Fuzzing OpenSSL

Brian Chen, Ashley Kim, Jason Lam

May 2019

1 Introduction

Fuzzing is a technique for testing the security of a program by programmatically providing it with many inputs, which may or may not be valid, and seeing if the program crashes or exhibits other insecure behavior. Our project concerns fuzzing OpenSSL, a standard open-source cryptography library used to secure online communications, with one of the most popular implementations of SSL and TLS. Among other uses, it is used to generate certificates for Let's Encrypt, an automated certificate authority that provides free SSL/TLS certificates.

Our main goal of this project was to successfully fuzz OpenSSL — that is, find a novel input that causes a crash, or other unexpected behavior, through automated testing or input generation. Through the course of our investigation, we found that there is already substantial computation power dedicated to fuzzing OpenSSL, while there was more available room for improvement in fuzzing infrastructure instead, so we pivoted to focusing on that. Our main contributions are improvements to fuzzing documentation and code coverage through writing additional fuzz targets.

2 Background and Previous Work

OpenSSL is an extremely popular open-source cryptography and SSL/TLS library first released in 1998. It secures many internet connections and provides many cryptographic primitives for other uses. Despite (or perhaps because of) OpenSSL's prevalence in being used to secure the modern internet, vulnerabilities in OpenSSL are frequently discovered. One of the most famous vulnerabilities in OpenSSL (and in modern software in general) is Heartbleed, a 2014 overrun vulnerability caused by a missing bounds check that allowed attackers to read additional data, and which was estimated to affect half a million certificates [2]. However, dozens of vulnerabilities each year are documented on OpenSSL's site [3]. Several of these vulnerabilities were found using the security testing technique of fuzzing, including:

- CVE-2018-0739: Malicious recursive ASN.1 types can crash OpenSSL.
- CVE-2017-3738: Overflow bug in AVX2 Montgomery multiplication.

- CVE-2017-3736: Carry propagating bug in the `x86_64` Montgomery squaring procedure.
- CVE-2017-3735: One-byte overread in parsing the `IPAddressFamily` extension of an X.509 certificate.
- CVE-2017-3732: Another carry propagating bug in the `x86_64` Montgomery squaring procedure.

In addition, although Heartbleed was not originally discovered through fuzzing, several tutorials have been written after its discovery about how it could have been detected with fuzzing [11, 12].

Fuzzing has become more common and more accessible in recent years. Bindings for parts of OpenSSL to popular fuzzers libFuzzer and AFL exist in the OpenSSL repository at `https://github.com/openssl/openssl/tree/master/fuzz`. Furthermore, many of the recent vulnerabilities have been found using Google's open source continuous fuzzer OSS-Fuzz [6]. OSS-Fuzz is backed by Google's distributed fuzzing infrastructure ClusterFuzz [17], which provides a friendly web interface and automated crash reports.

3 Fuzzing Overview

In this section we cover how fuzzers work, including how they integrate with target software, generate inputs, and detect potential vulnerabilities. We also discuss the fuzzing engines we explored during our project.

3.1 Integrating Fuzzers with Targets

How a fuzzer is integrated into software varies from fuzzer to fuzzer, but typically, it requires the software implementer to implement a *fuzz target*, which is a function that simply takes in a raw array of bytes and somehow processes it with the software. The fuzz target should be written so that it never crashes on any data if the library is operating correctly; this might mean carefully checking the received data for validity before processing it [5].

It is good practice to declare many small fuzz targets and have them be fuzzed independently, as it enables the fuzzer to test targets more precosely. Some examples of existing fuzz targets in OpenSSL are functions that treat the input data as bignums, perform simple arithmetic operations on them, and check the results for consistency with each other. Other targets treat the data as part of an SSL connection between the server and client.

Sometimes, fuzz targets can be written to be supported by multiple fuzzing engines.

3.2 Running Fuzzers

Once the fuzzer and target have been integrated, the fuzzer typically generates inputs by applying randomness, especially genetic or evolutionary algorithms,

to a *corpus* of inputs, which is initially user-provided.

The corpus ideally consists of inputs that, in loose terms, do interesting things, with the goal of helping the fuzzer to cover as much of the code or behavior as possible. This helps fuzzers test interesting code paths quickly. For example, if a code path is only reached if a message has a correct cryptographic signature, it would help the fuzzer if the corpus included an input with a message and its correct signature, because the fuzzer would be unlikely to generate such an input by itself, and would then be unable to test that code path. Once the corpus contains at least one such input, however, the fuzzer can efficiently find and test similar inputs (e.g. inputs with extra padding) and plausibly find bugs or vulnerabilities in that code path. As fuzzers run, they may add inputs to the corpus when they discover an input that triggers a new code path.

OpenSSL's corpora are included in their repo [4], simply as directories of files named by hashes of their contents, and feature varying inputs as described above. For example, the corpora for the bignum tests include many random numbers of various sizes, but also edge cases like when all numbers are 0; the corpora for the SSL connection targets include random data as well as transcripts that include legitimate SSL certificates from Google. When fuzz targets can be declared in a fuzzer-agnostic way, a benefit is that corpora can be exchanged between different fuzzers, which improves the ability of fuzzers to find new code paths.

3.3 Sanitizers

Fuzzer effectiveness can be further improved by fuzzing code compiled with *sanitizers*, which are a type of compiler instrumentation that turns potentially vulnerable code behavior into fatal errors. Examples of behavior that might be caught by sanitizers include out-of-bounds memory accesses (by AddressSanitizer [18]), memory leaks (by LeakSanitizer [19]), reads from uninitialized memory (by MemorySanitizer [20]), and undefined behavior (by UBSanitizer [21]).

Although sanitizers improve the ability of fuzzers to find issues, they are typically only used for fuzzing and similar debugging or evaluation purposes because the added instrumentation usually makes the compiled code slower or more memory-consuming.

3.4 Other Fuzzing Conditions

Finally, effective fuzzing sometimes requires fuzz targets to be modified. For example, fuzzing is the most effective when fuzz targets are fully deterministic and do not rely on any global state or randomness, because this makes it more likely that similar fuzz inputs will explore similar code paths and ensures that fuzz failures are reproducible. This often requires the software target be modified to support a derandomized build. Of course, a cryptography library without randomness is extremely vulnerable and should never be used in production, and similar concerns apply to many other modifications one

might wish to make to support fuzzing. To drive this home, libfuzzer officially suggests that such modifications be gated behind the long build macro `FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION` [5].

However, although OpenSSL's fuzzing documentation suggests that this build macro disables randomness in two tests, we found that it does not actually appear to affect randomness at all; instead, randomness is explicitly disabled in the two relevant fuzz tests that require determinism.

3.5 libfuzzer

libfuzzer is part of `clang`, a compiler suite that is in turn part of the LLVM project. It works by linking a custom main function against fuzz targets that are simply functions declared with the following exact signature and name [5]:

```
1 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
2     // do something with the data buffer
3     return 0;
4 }
```

To fuzz a simple target with a modern version of `clang`, it suffices to compile it with the `-fsanitize=fuzzer` option; this adds all the necessary instrumentation and links a custom libfuzzer `main` function against the fuzz target function. `clang` also ships with many sanitizers that can detect runtime behavior that could cause security issues, which are typically also enabled during fuzzing.

However, in a library such as OpenSSL, attempting to do this directly will fail because libfuzzer's main function will conflict with OpenSSL's own main function, used for the `openssl` binary. Modern `clang` also accepts a `-fsanitize=fuzzer-no-link` option that will only add fuzz instrumentation without linking, but of course the fuzzer main method needs to be linked eventually. Currently, the OpenSSL build process handles this in a somewhat roundabout way with a `--with-fuzzer-lib` configuration option that takes a path to the libfuzzer library file, which was necessary before libfuzzer was shipped with clang.

While these technical details with building/compilation seem like technical minutiae, their complexity has consequences. For about a month while we were working on this project, the build of OpenSSL on oss-fuzz (described below) was failing due to a linking issue with the fuzzing library and the OpenSSL makefile [1]. OpenSSL maintainers eventually resolved the issue with pull request #8892 [24].

3.6 AFL

American fuzzy lop is an another major security fuzzing engine we used. AFL must be installed separately from `clang`, and ships with several binaries including `afl-clang` and `afl-clang-fast`. These binaries can be used as drop-in replacements for `clang` during compilation; the resulting program will be ready for fuzzing using the `afl-fuzz` program.

One reason that AFL runs quickly is due to its persistent mode [16]. Most general-purpose fuzzers create a new process for each new input, but creating a new process is quite slow. While an approach known as the forkserver optimization exists, where the fuzzer saves some time calling `fork()` instead of `execve()` on each iteration, this is still much slower than a basic for loop in C.

AFL introduces an `__AFL_LOOP()` macro that essentially works the same as a for loop. While there is still crash handling, stall detection, and instrumentation being handled by AFL under the hood, this approach still ends up being faster than making a `fork()` or `execve()` call on every new iteration.

3.7 Other Fuzzers

The list of fuzzers is by no means exhaustive. Some other fuzzers include `honggfuzz` [8], `radamsa` [9], and `zzuf` [10].

3.8 OSS-Fuzz

OSS-Fuzz by Google is a platform that continuously fuzzes open-source software, supporting the two fuzzers primarily discussed above. The code is available at `https://github.com/google/oss-fuzz`. OpenSSL is one of the dozens of open-source projects being continuously fuzzed on OSS-Fuzz. Because Google has so many computing resources, we decided that it would be unlikely for us to find any issues by running the same fuzzing processes on our own machines. On the other hand, as mentioned above, we did observe from the public build logs [13] that for some amount of time, the OpenSSL build on OSS-Fuzz was not building.

A different area of potential work can be seen from the public coverage reports [7]. (They are referenced as public in the OSS-Fuzz FAQ [15]. However, we could not find anywhere they were publicly indexed, and only managed to find them through finding links to other OSS-Fuzz coverage reports and modifying the URL.)

It can be seen that the fuzz coverage of OpenSSL is rather low, just around 38%. Although full fuzzer coverage is not necessarily realistic or desirable, since some functions are difficult to incorporate into a reliable fuzz target, we think this percentage suggests that there is a lot of room for improvement.

However, the actual status and results of fuzzing are maintained on a separate bugtracker [14], and permission-gated to only project maintainers. This is important because it gives maintainers a chance to fix any vulnerabilities uncovered by fuzzing before they are made public, reducing the chance that the vulnerabilities are maliciously exploited. Unfortunately, it also means we cannot use Google's fuzzing results to cross-check or guide our own fuzzing attempts.

4 Our Contributions

4.1 Documentation

While attempting to fuzz OpenSSL, we quickly discovered that the existing documentation for fuzzing OpenSSL was out of date. Following the instructions for compiling OpenSSL with libfuzzer didn't work with the current version of `clang`, so we had to read a lot of documentation and do some experimentation to figure out how to build the new binary. While our first successful attempts involved monkey-patching the autogenerated OpenSSL makefile to use `-fsanitize=fuzzer` in some places and `-fsanitize=fuzzer-no-link` in others, it took several days before we figured out how to compile the fuzz targets leveraging both OpenSSL's configuration scripts as well as modern `clang` flags. (We later found that OSS-Fuzz is also attempting to migrate to use modern `clang` and its prebuilt fuzzer libraries and running into similar issues with the two fuzzing flags [25].) What we ended up doing was significantly informed by observing the build commands issued to build OpenSSL in the OSS-Fuzz docker image, which were constructed through a complex combination of code in the OpenSSL build files and various OSS-Fuzz infrastructural scripts.

After getting a working build with libfuzzer, we submitted a pull request to update the fuzzer readme on OpenSSL to reflect the new steps. The pull request was approved on May 8^{th} [23].

We had relatively more success with AFL, as we only encountered a few hiccups regarding in the installation of `afl-clang-fast`. After emailing the `afl-users` Google Group, we got help and our compilation errors were resolved and `afl-fuzz` ran without issue.

4.2 Vulnerable Inputs

While running the pre-existing libfuzzer fuzzers, we found several slow inputs. In theory, slow inputs could reflect a vulnerability to denial-of-service attacks (via overwhelming an OpenSSL server or client with inputs that it needs a long time to process) or side-channel timing attacks (via measuring how long an OpenSSL server or client takes to respond to a certain input to leak a secret). In practice, we believe that the slow inputs we found simply reflect the fact that it takes more time to process large inputs, and that there are no actual security vulnerabilities. Also, the generation of these large inputs can be disabled with proper fuzzer flags and may be set up correctly on OSS-Fuzz. Nevertheless, the presence of slow inputs may at the very least reduce the efficiency and likelihood of using fuzzing to discovering other legitimate bugs and vulnerabilities.

When fuzzing with AFL, we never ended up encountering any program crashes or hangs.

4.3 Coverage

While looking at OSS-Fuzz's documentation, we found the Google's coverage report for OpenSSL fuzzing [7].

We noticed that there were plenty of code paths and function calls that had low coverage, so we decided to write a few fuzz targets ourselves.

We wrote new fuzz targets for the Diffie-Hellman key exchange, the `gcd` function for `BigNumber`s, and the `hex2bn` and `dec2bn` deserialization functions for `BigNumber`s. When we ran our own tests, we found higher coverage rates for our targeted functions than OSS-Fuzz's previous results. The source code for these fuzz targets is available in the appendix.

The last 3 functions were pretty straightforward, but testing the functions for Diffie-Hellman required a bit more work. We needed to go through the entire key exchange procedure with only one machine, and there was not a dedicated API for generating private keys that depended on input, so we had to use fairly low-level API functions and manually inject `BigNumber`s parsed from the input into the key exchange.

5 Further Work

There is a lot more work that could be done in fuzzing OpenSSL. There are several files with very little to no coverage that may potentially be vulnerable. For example, the RSA module has gotten relatively low coverage in the last OSS-Fuzz report, with under 35% coverage. The X.509 certificate module has also very low coverage, with less than 30% of its lines covered. More fuzz targets can be written to increase fuzzing coverage in these files.

Furthermore, instead of simply relying on blackbox fuzzing, we could use additional techniques to further test the system for robustness.

A common whitebox fuzzing technique is symbolic execution, which replaces the semantics of the program with symbolic ones, and ensures that all possible execution paths are traversed. However, a library as large as OpenSSL has too many constraints, making it infeasible to symbolically execute in a reasonable amount of time.

A technique introduced around 2015 called underconstrained symbolic execution remedies this problem. It relies on checking individual functions instead of whole programs for correctness. Recently, the security company Trail of Bits released Sandshrew, a underconstrained symbolic execution tool explicitly for use in cryptographic analysis [22]. This could be used for more robust testing on OpenSSL.

6 Conclusion

With the amount of computing power that Google is contributing towards fuzzing OpenSSL through OSS-Fuzz, it was not surprising that we could not find any new crashes or vulnerabilities in OpenSSL. Nonetheless, we were able

to contribute to OpenSSL by updating outdated documentation and writing new fuzz targets to run. We hope that fuzzing in OpenSSL and in all security-sensitive software continues to improve, and that it will be able to discover vulnerabilities more quickly.

References

[1] Issue 14075, oss-fuzz, "openssl: Build failure". https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=14075

[2] Paul Mutton, "Half a million widely trusted websites vulnerable to Heartbleed bug." Netcraft.com. https://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html

[3] OpenSSL Vulnerabilities https://www.openssl.org/news/vulnerabilities.html

[4] https://github.com/openssl/openssl/tree/master/fuzz/corpora

[5] http://llvm.org/docs/LibFuzzer.html

[6] OSS-Fuzz by Google https://github.com/google/oss-fuzz

[7] https://storage.googleapis.com/oss-fuzz-coverage/openssl/reports/20190512/linux/report.html. (Note that reports are published daily and the date in the URL can be manually changed.)

[8] honggfuzz http://honggfuzz.com

[9] radamsa https://gitlab.com/akihe/radamsa

[10] zzuf https://caca.zoy.org/wiki/zzuf

[11] https://google.github.io/clusterfuzz/setting-up-fuzzing/heartbleed-example/

[12] Hanno Böck, "How Heartbleed could've been found." 2015. https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

[13] https://oss-fuzz-build-logs.storage.googleapis.com/index.html

[14] https://bugs.chromium.org/p/oss-fuzz/issues/list

[15] https://github.com/google/oss-fuzz/blob/master/docs/faq.md#why-are-code-coverage-reports-public

[16] lcamtuf, "New in AFL: persistent mode." https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html

[17] https://google.github.io/clusterfuzz/

[18] AdressSanitizer http://clang.llvm.org/docs/AddressSanitizer.html

[19] LeakSanitizer http://clang.llvm.org/docs/LeakSanitizer.html

[20] MemorySanitizer http://clang.llvm.org/docs/MemorySanitizer.html

[21] UndefinedBehaviorSanitizer http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[22] Alan Cao, "Performing Concolic Execution on Cryptographic Primitives", Trail of Bits Blog. https://blog.trailofbits.com/2019/04/01/performing-concolic-execution-on-cryptographic-primitives/

[23] OpenSSL Pull Request #8891 https://github.com/openssl/openssl/pull/8891

[24] OpenSSL Pull Request #8892 https://github.com/openssl/openssl/pull/8892

[25] OSS-Fuzz Issue #2164 https://github.com/google/oss-fuzz/issues/2164

A Fuzzer Targets

A.1 dh.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <openssl/bn.h>
4  #include <openssl/err.h>
5  #include <openssl/dh.h>
6  #include "fuzzer.h"
7
8  int FuzzerInitialize(int *argc, char ***argv)
9  {
10     // OPENSSL_init_crypto(OPENSSL_INIT_LOAD_CRYPTO_STRINGS, NULL);
11     // ERR_get_state();
12
13     return 1;
14 }
15
16 int FuzzerTestOneInput(const uint8_t *buf, size_t len)
17 {
18     BN_CTX *ctx = BN_CTX_new();
19     size_t l1 = 0;
20     BIGNUM *b1 = BN_new();
21     BIGNUM *b2 = BN_new();
22
23     /* Extract two bignums from the input by dividing the input
           into two parts,
24      * using the values of the first byte to choose lengths.
```

```
25          */
26      if (len >= 1) {
27          len -= 1;
28          l1 = (buf[0] * len) / 255;
29      }
30
31      OPENSSL_assert(BN_bin2bn(buf, l1, b1) == b1);
32      OPENSSL_assert(BN_bin2bn(buf + l1, len - l1, b2) == b2);
33
34
35      DH *privkey1 = DH_get_2048_256();
36      DH *privkey2 = DH_get_2048_256();
37      int codes;
38      OPENSSL_assert(1 == DH_check(privkey1, &codes));
39      OPENSSL_assert(codes == 0); // problems with the generated
            parameters
40      OPENSSL_assert(1 == DH_check(privkey2, &codes));
41      OPENSSL_assert(codes == 0); // problems with the generated
            parameters
42
43      // Set private keys manually
44      BIGNUM *p1 = BN_new();
45      BN_mod(p1, b1, DH_get0_q(privkey1), ctx);
46      BIGNUM *p2 = BN_new();
47      BN_mod(p2, b2, DH_get0_q(privkey2), ctx);
48
49      // They have to be at least 2
50      if (!BN_is_zero(p1) && !BN_is_one(p1) &&
51              !BN_is_zero(p2) && !BN_is_one(p2)) {
52          DH_set0_key(privkey1, NULL, b1); // note that this
                transfers memory management to the DH keys
53          DH_set0_key(privkey2, NULL, b2); // ditto
54          OPENSSL_assert(1 == DH_generate_key(privkey1));
55          OPENSSL_assert(1 == DH_generate_key(privkey2));
56          unsigned char *shared1, *shared2;
57          size_t shared_size1, shared_size2;
58
59          OPENSSL_assert(NULL != (shared1 =
                OPENSSL_malloc(sizeof(unsigned char) *
                (DH_size(privkey1)))));
60          OPENSSL_assert(NULL != (shared2 =
                OPENSSL_malloc(sizeof(unsigned char) *
                (DH_size(privkey2)))));
61          OPENSSL_assert(0 <= (shared_size1 =
                DH_compute_key(shared1, DH_get0_pub_key(privkey2),
                privkey1)));
62          OPENSSL_assert(0 <= (shared_size2 =
                DH_compute_key(shared2, DH_get0_pub_key(privkey1),
                privkey2)));
63          OPENSSL_assert(shared_size1 == shared_size2);
64          OPENSSL_assert(memcmp(shared1, shared2, shared_size1) ==
                0);
65          OPENSSL_free(shared1);
66          OPENSSL_free(shared2);
67      } else {
68          BN_free(p1);
69          BN_free(p2);
```

```
70      }
71
72      BN_free(b1);
73      BN_free(b2);
74      DH_free(privkey1);
75      DH_free(privkey2);
76      BN_CTX_free(ctx);
77      ERR_clear_error();
78
79      return 0;
80  }
81
82  void FuzzerCleanup(void)
83  {
84  }
```

A.2 gcd.c

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <openssl/bn.h>
4   #include <openssl/err.h>
5   #include "fuzzer.h"
6
7
8   int FuzzerInitialize(int *argc, char ***argv)
9   {
10      // OPENSSL_init_crypto(OPENSSL_INIT_LOAD_CRYPTO_STRINGS, NULL);
11      // ERR_get_state();
12
13      return 1;
14  }
15
16  int FuzzerTestOneInput(const uint8_t *buf, size_t len)
17  {
18      BN_CTX *ctx = BN_CTX_new();
19      size_t l1 = 0;
20      BIGNUM *b1 = BN_new();
21      BIGNUM *b2 = BN_new();
22      BIGNUM *gcd = BN_new();
23      BIGNUM *gcd_rel_prime = BN_new();
24
25      // b1 = gcd * d1, b2 = gcd * d2
26      BIGNUM *d1 = BN_new();
27      BIGNUM *d2 = BN_new();
28      // remainders
29      BIGNUM *r1 = BN_new();
30      BIGNUM *r2 = BN_new();
31
32
33
34      /* Extract two bignums from the input by dividing the input
            into two parts,
35       * using the values of the first byte to choose lengths.
36       */
37      if (len >= 1) {
38          len -= 1;
```

```
39          l1 = (buf[0] * len) / 255;
40      }
41
42      OPENSSL_assert(BN_bin2bn(buf, l1, b1) == b1);
43      OPENSSL_assert(BN_bin2bn(buf + l1, len - l1, b2) == b2);
44
45      // We want the GCD to be nonzero
46      if (!BN_is_zero(b1) && !BN_is_zero(b2)) {
47          BN_gcd(gcd, b1, b2, ctx);
48          BN_div(d1, r1, b1, gcd, ctx);
49          BN_div(d2, r2, b2, gcd, ctx);
50
51          OPENSSL_assert(BN_is_zero(r1));
52          OPENSSL_assert(BN_is_zero(r2));
53
54          BN_gcd(gcd_rel_prime, d1, d2, ctx);
55          OPENSSL_assert(BN_is_one(gcd_rel_prime));
56      }
57
58      BN_CTX_free(ctx);
59      BN_free(b1);
60      BN_free(b2);
61      BN_free(gcd);
62      BN_free(gcd_rel_prime);
63      BN_free(d1);
64      BN_free(d2);
65      BN_free(r1);
66      BN_free(r2);
67      ERR_clear_error();
68
69      return 0;
70  }
71
72  void FuzzerCleanup(void)
73  {
74  }
```

A.3 bnprint.c

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <openssl/bn.h>
4   #include <openssl/err.h>
5   #include <openssl/dh.h>
6   #include "fuzzer.h"
7
8
9   int FuzzerInitialize(int *argc, char ***argv)
10  {
11      return 1;
12  }
13
14  int FuzzerTestOneInput(const uint8_t *buf, size_t len)
15  {
16      BN_CTX *ctx = BN_CTX_new();
17      size_t l = 0;
18      BIGNUM *b1 = BN_new();
```

```
19      BIGNUM *b2 = BN_new();
20      BIGNUM *b3 = BN_new();
21
22      if (len >= 1) {
23          len -= 1;
24          l = (buf[0] * len) / 255;
25      }
26
27      OPENSSL_assert(BN_bin2bn(buf, l, b1) == b1);
28      char* buf2 = BN_bn2hex(b1);
29      char* buf3 = BN_bn2dec(b1);
30      OPENSSL_assert(BN_hex2bn(b2, buf2) >= 0);
31      OPENSSL_assert(BN_dec2bn(b3, buf3) >= 0);
32
33      OPENSSL_free(buf2);
34      OPENSSL_free(buf3);
35      BN_free(b1);
36      BN_free(b2);
37      BN_free(b3);
38      BN_CTX_free(ctx);
39      ERR_clear_error();
40
41      return 0;
42  }
43
44  void FuzzerCleanup(void)
45  {
46  }
```