Security of OpenSSL Tool

John Phillips
johphill@mines.edu

2nd Abhaya Shrestha
ashrestha@mines.edu

3rd Joe Granmoe
jgranmoe@mines.edu

4th Patrick Curran
pcurran@mines.edu

5th Collin McDade
collinmcdade@mines.edu

6th Noor Malik
nmalik@mines.edu

Abstract—There have been small but catastrophic modifications to openssl in the past [1] [2] [5] that cause massive security problems and in some cases exploits that can disclose private keys. We explore in depth one such instance, a modification made to the OpenSSL package distributed in debian that resulted in only 32,768 possible keys that could be generated. We also explore similarities to heartbleed. Finally, we investigate methods that have been put in place to prevent such bugs from happening in the future.

Index Terms—OpenSSL, security vulnerability, debian

I. INTRODUCTION

II. BACKGROUND

In 2008 a debian developer made a patch to openssl to fix valgrind warnings [2] [3]. OpenSSL has a flexible system for generating random numbers that uses an interface-like approach with functions prefixed with `RAND_*` delegating to a particular implementation. Since this API was designed for cryptographic applications, the random bytes generated need to not just satisfy particular statistical properties but be truly unpredictable. In order to acheive this property, this particular random number generator allows the user to add 'entropy' by feeding it buffers full of random bytes. These bytes could be from `/dev/urandom`, the user, or somewhere else. When adding entropy in this way to the message digest-based random API, an internal hash state is update each time with the random bytes given by the user. The hash state then can be used to get random bytes back out of the system.

Now this particular implementation in `md_rand.c`, *deliberately* would 'over-read' the user-provided buffer. The apparent thinking behind this was that if the code read uninitialized memory, this couldn't hurt things, and could only possibly help by potentially adding randomness.

This behavior occurred in two places in `md_rand.c`. Both of those places were actually critical to the entire random number generator. Because of the reading of uninitialized memory, any time a user of the ssl library did almost anything – like generate a key, for example – they would get disturbing warnings if they ran their program in the popular `valgrind` tool.

Most linux distributions maintain small set of 'patches' for each package they distribute. These patches normally are designed to fix critical bugs that haven't been fixed upstream or are in the process of being fixed. So although the debian project doesn't maintain OpenSSL, they could patch the particular version of OpenSSL distributed by debian. And that

was the solution arrived at in the bug ticket and over email [2] [4].

This was the source of a complaint that first came to the debian bug tracker [2]. The debian developer quickly spotted the culprit code but didn't know what it did. Observing that commenting it out didn't cause anything to crash or obviously function incorrectly, he asked on the openssl development mailing list if it was okay to remove the two lines of code – and got the go-ahead.

The debian developer then applied the patch and distributed the debian version of OpenSSL with this critical vulnerability built in. This vulnerability was distributed to all users of debian from 2006 to 2008 [1]. During that entire time, any code using the random number generator could only really have 32,767 possible outputs from the random number generator. Examples of affected functions included TLS modules and SSH key generation – actually, anything that generated RSA keys was vulnerable.

III. RECREATING THE 2008 BUG

In order to gain a better understanding of exactly how the bug worked, and how easy (or difficult) it would be to exploit such a flaw, we used virtual machines and code from the 2008 era to see what exploiting the vulnerability might look like. The basic idea was to create two virtual machines. One would be the exploited machine – a stand-in for a server on the Internet or other network that had an open SSH port that was believed to be secure. The other machine would try to access the target server over SSH by repeatedly generating trial keys until it was able to successfully get in. Although this is not as simple as just writing code to guess the key and compares two files, it is arguably a more realistic demonstration of how exploits really work, and a better way of understanding them, putting the 'applied' in 'applied cryptography'.

A. Code modifications

First, we created two different versions of the OpenSSL code. The first one had the two lines that the debian developer asked about on the OpenSSL mailling list commented out (see [4]), and nothing more. It was a re-creation of what existed in the debian project from 2006-2008. We looked for the exact code (for example, a .deb binary distribution, or the actual patchsets applied to the OpenSSL project) on the Internet but were unable to find it (links pointing to the patches were all

```
List:       openssl-dev
Subject:    Re: Random number generator, uninitialised data and valgrind.
From:       Ulf_Möller <ulf () openssl ! org>
Date:       2006-05-01 22:34:12
Message-ID: 44568CE4.9020906 () openssl ! org
[Download RAW message or body]

Kurt Roeckx schrieb:
> What I currently see as best option is to actually comment out
> those 2 lines of code.  But I have no idea what effect this
> really has on the RNG.  The only effect I see is that the pool
> might receive less entropy.  But on the other hand, I'm not even
> sure how much entropy some unitialised data has.
>
Not much. If it helps with debugging, I'm in favor of removing them.
(However the last time I checked, valgrind reported thousands of bogus
error messages. Has that situation gotten better?)
```

Fig. 1. A reply to the email from the Debian Developer telling him to go ahead and modify the OpenSSL code. [4]

dead, possibly in a deliberate attempt by debian to prevent anyone accidentally using the code again).

The next branch of the OpenSSL code was similar to the first except it had a way to manually set the only real source of variablility left to the random number generator – the PID (process ID). We did this by replacing calls to getpid() in md_rand.c to references to a static variable in md_rand.c called fake_pid. We also introduced a globally-visible function called RAND_exploit_set_pid(), to allow client code to set the PID.

B. Virtual machine setup

We built two virtual machines based on debian 5. The machines were connected to a single virtual switch, and had static IPs assigned. On one virtual machine, we built and installed the re-created debian-distributed OpenSSL. We used ssh-keygen to create and authorize a key as an authorized SSH key on that host.

On the other (attacker) machine, we used a program derived from ssh-keygen to try to SSH to the target machine with a given PID seed. That program then was repeatedly invoked by another script that iterated over all possible PIDs, from 1 to 32,768 (in 2008 that was the number of possible PIDs).

The results were as expected – the attacker machine was able to generate the public/private key pair of the target machine within fifteen to twenty minutes. On a fresh boot, because there haven't been as many processes run, it takes only about five minutes.

C. Running the exploit

Since the virtual machines are connected over a virtual, network, to run the exploit, we simply boot them and run a script on the attacker machine that will eventually break the key of the target machine. The attacker script will then write the public/private key pair to a file. See the appendix for details.

D. Alternative approaches

IV. SIMILARITIES TO HEARTBLEED

The debian bug bears certain similarities to another bug that occurred in 2014 – Heartbleed [4]. Both bugs compromised private keys, and both were not the result of any protocol design errors, cryptanalytic breakthroughs that allowed algorithms to be broken, or any other issue commonly learned about in a cryptography class. Instead, both errors were the result of human error. Simple, and (in hindsight) rather obvious bugs made their way into OpenSSL packages that were distributed to an untold number of machines.

With heartbleed, the issue was simply a missing boundary check¡CITE¿. First, code was added to allow for an extension to the SSL/TLS protocol that sent 'heartbeats'. This allows for the SSL connection to be kept alive ¡CITE¿. The fix was only a few lines of code¡CITE¿.

Although one bug was an alteration of working code and the other was adding a buggy feature, the critical piece of both involved reading past array bounds. In the debian bug, a call to MD_Update() was reading past initialized memory into unused memory and caused a valgrind error, which triggered the faulty fix. In Heartbleed, a boundary check was not made that allowed attackers to read uninitialized memory.

This raises the question of how, if possible, to catch such simple implementation errors? Possible solutions are using CI/CD and a good automated test suite that has high code coverage and covers edge-cases. We can see that both bugs would have been stopped if:

- The OpenSSL library was designed from the start to use static and dynamic analysis tools like valgrind (no false positives as in the case of the debian bug, and possible warnings in the case of Heartbleed), sanitizers, and static code analysis tools like cppcheck.

- The OpenSSL library had high-coverage functional testing, and a test suite that was automatically run before anyone distributed a new or modified OpenSSL. This can be accomplished with compiler tools that generate test coverage data like `gcov` and `lcov`.

In the next section we examine if this situation has improved.

V. INVESTIGATING THE CURRENT CODE

A. Architectural differences

In the current code, the old `md_rand.c` has been removed. It was renamed to `ossl_rand.c` in June of 2017 [13], and removed in August, to be replaced by the DRBG api [14]. The DRBG api is based on the NIST standard NIST SP 800-90A, for 'Deterministic Random Number Generator' [10] [15].

The current code also has a much greater unit test coverage in the `test/` directory. Where previously there were a few very simple and esoteric tests based on shell scripts to check basic correctness – for example, the file `test/trsa` is a shell script that tests if the `openssl rsa` command can convert RSA keys between different plaintext formats – there are now much more comprehensive tests that test the DRBG for duplicate results. If these had been in place before the debian key bug, perhaps it never would have happened.

B. Code quality improvements

We were able to demonstrate that the valgrind errors observed in [2] actually were visible in the unit tests themselves in the old OpenSSL (version `0.9.7e`), when running the test `exptest` in the `test` directory (see Appendix A for details). Not only that but the OpenSSL code in general from that era was littered with valgrind errors, with valgrind not counting more errors on some tests because there were so many. Using the `lcov` tool, code coverage was measured at around 42%, covered in Appendix A. This left lots of room for bugs to slip through, and none of the tests failed on the branch that contained the debian modifications – so the debian developer had no way to test his work other than trust in the mailing list.

The current state of the code (the `master` branch as of March 25, 2022) is certainly better, although perhaps by not as much as would be ideal. We ran the functional test suite in the `test/` directory to see code coverage, using the same method for `0.9.7e`. Code coverage improved to a total of 66% of lines (see Appendix A for how to get this information for yourself). We noticed that some of the implementation files for the DRBG, in `providers/implementations/rands/drbg.c` had many uncovered edge-cases. Although the code may work now, uncovered edge cases in the test suite mean that the code isn't defended from potential future changes like the one made by the debian developer in 2006.

We did confirm that similar alternations to the DRBG code to the ones made in 2006 by the debian developer *did* cause unit tests to fail in the newer test suite (version 3.0.0).

VI. PROCESSES IN THE OPENSSL PROJECT IN PLACE TO PREVENT FURTHER EXPLOITS

Exploring OpenSSL project contribution process, we looked into what it takes to be a committer in their repo, whether they have any kind of automated checking in place (CI or Continuous Integration), and what their approval process looks like.

Before we delve into the process, we need to define a few terms from their glossary.

1) **OMC (OpenSSL Management Committee)**: they oversee all managerial and administrative aspects of the project. They are the final authority for the project [6]
2) **OTC (OpenSSL Technical Committee)**: they oversee all technical aspects of the project [6].

A. What it takes to be a Commiter of OpenSSL?

To become a committer, one has to be granted access by the OMC on the recommendation of OTC [7]. This access can be withdrawn at any time by a vote of the OMC [8]. In order to maintain their committer status, a commiter must have authored or reviewed at least one commit within the previous two calendar quarters [8].

They expect committers to be experts in some part of low-level crypto library as well as generalists who contribute to all areas of codebase [7]. Committers are expected to oversee the health of the project: fixing bugs, addressing open issues, reviewing contributions, and improving tests and documentation [7]. According to OpenSSL, to be a commiter, one can start by contributing code, reading code style, and getting to know build and test system [7]. In short, anyone can be a commiter as long as they are granted access by OMC on recommendation of OTC. Their contributions can range from small documentation fixes, spelling errors, to large code base contributions.

B. What automation checks does OpenSSL have to ensure code safety?

Based off of their glossary, we were able to understand that OpenSSL uses Continuous Integration which is a suite of tests and checks that are run on every pull request, commit on a daily basis. Some of their tests include functional tests, performance testing, and fuzz testing [9]. These kinds of tests are described as follows.

1) **Functional Tests**: given a set of inputs, it will produce a correct set of outputs [9].
2) **Performance Tests**: These tests test performance, and will be performed automatically via CI on a regular basis for certain components. Examples include:
   - Individual algorithm performance operating over different input sizes
   - SSL/TLS handshake time over multiple handshakes and for different protocol versions and resumption/non-resumption handshake.

   [9].

3) **Fuzz Tests**: Systematic methodology that is used to find buffer overruns (remote execution); unhandled exceptions, read access violations (AVs), and thread hangs (permanent denial-of-service), and memory spikes (temporary denial-of-service) [10].

According to their testing policy, a pull request does not require testing if the changes are as follows [9].

- Documentation (including CHANGES/NEWS file changes)
- The test suite
- perl utilities
- include files
- build system
- demos

Additionally, they recommend adding fuzz testing when implementing significant new functional tests or at least consider adding fuzz testing when refactoring to check all corner cases are covered [9]. OpenSSL particulary uses either LibFuzzer or AFL to do fuzz testing [11]. While performance tests improve the functionality of OpenSSL, fuzz testing and functional testing helped us ensure that OpenSSL codebase is at least sanity checked. Fuzz tests especially help since they are basically dynamic analysis detection for any vulnerabilities or bugs in the codebase.

C. What does their approval and code review process look like?

OpenSSL approval and code reviews process are as follows.

- OpenSSL pull requests are reviewed and approved by at least two committers, one of whom must be an OTC member. Neither of the reviewers can be the author of the submission [7]. Only exception to this policy is during the release process when the author's review does count towards the two needed reviews for the automated release process and NEWS and CHANGES file updates [7].
- In case two committers make a joint submission, they can review each other's code, but not their own. Additionally, a third reviewer will be required [7].
- An OMC/OTC member may apply a needs OMC decision label to a submission which can hold the submission. This hold may only be removed by OTC/OMC member who put the hold or the OTC/OMC [7].
- Approved submission (besides automated release process and NEWS and CHANGES file updates) will only be applied after 24-hour delay from approval. An exception to the delay exists for build and test breakage fix approvals which will be flagged with the severity: urgent label [7].

Additionally, on their commit workflow, their public github repository is just a mirror and pulls from the real repo where only committers have access [7] (When someone becomes a committer, OpenSSL will send instructions to get commit access to the repository [7]). Finally, they also have strict documentation design process whenever minor or major code changes the design, and changes to existing public API functions and data are forbidden [9]. However, they encourage a new API call to be implemented if need be as a minor release [9]. From this information, we can definitely tell that today it is difficult to be able to commit anything without prior granted permissions and approval to OpenSSL. Futhermore, even if changes were to occur, they always run their automation process to detect any defects or vulnerabilities with their test suites which is very assuring for a user of OpenSSL.

However, there are still chances that a defect or vulnerability could be shipped out. Particularly, when they suggested that refactors do not need a test but recommend Fuzz testing to cover edge cases. In the refactor case, there is a possibility that the committer misses an edge case in fuzz testing and creates a bug or vulnerability. Another challenge of a cryptographic tool such as OpenSSL is that professionals do not know how to test for security. An algorithm implementation is only secure if an attacker cannot decipher or find vulnerabilities to exploit for the algorithm. Futhermore, Fuzz tests are basically just shooting in the dark and hoping to find a vulnerability before the adversary finds it. Nonetheless, we have to say their process for who gets to commit code to the master is solid; however, there is no assurance that vulnerability will not be shipped out. The best we could do is hope that the code review and automation process detects a defect or vulerability and prevents it from being shipped out.

VII. Conclusion

To understand how a vulnerability like the 2008 debian OpenSSL bug happens and can be exploited, we read the code from that period and built an exploit that would be able to unlock affected machines using SSH keys.

Next, we did a survey of the state of the code then and now to see if the introduction of such code was due to incompetence on the debian developer's part, or an understandable error that no automated tools were in place to catch. We concluded that there was no easy way to tell if the code modification would adversely affect the security of the library, and that, furthermore, the debian developer was given the go-ahead.

The current state of the code is much better however, having more thorough unit-testing coverage that can catch more accidental errors made by OpenSSL developers or package distributors. Furthermore, the current implementation of OpenSSL's random number generator is based on a specification and is therefore much easier to reason about.

We concluded that the security of the OpenSSL tool has increased substantially since the 2008 debian OpenSSL bug was caught, the heartbleed vulnerability notwithstanding.

References

[1] https://isotoma.com/blog/2008/05/14/debians-openssl-disaster/
[2] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516
[3] https://research.swtch.com/openssl
[4] https://marc.info/?l=openssl-dev&m=114652287210110&w=2
[5] https://heartbleed.com/
[6] https://www.openssl.org/policies/glossary.html
[7] https://www.openssl.org/policies/general/committer-policy.html
[8] https://www.openssl.org/policies/omc-bylaws.html

[9] https://www.openssl.org/policies/technical/

[10] https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)?redirectedfrom=MSDN

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

[12] "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", revision 1, https://doi.org/10.6028/NIST.SP.800-90Ar1, June 2015.

[13] Salz, Richard, OpenSSL Source code, commit da8fc25a989cf4f4d26d626a85477e8a9282da12, June, 2017.

[14] Salz, Richard, OpenSSL Source code, commit 75e2c877650444fb829547bdb58d46eb1297bc1a, August 2017.

[15] OpenSSL Project, "RAND_DRBG", OpenSSL Documentation, https://www.openssl.org/docs/man1.1.1/man7/RAND_DRBG.html

APPENDIX A
BUILDING AND RUNNING THE EXPLOIT

For the project we built a git repository found at https://github.com/john-w-phillips/crypto-project. The file README.md there contains detailed information about running the exploit but we include some here as a general overview.

First, we'll cover alterations we made to the OpenSSL code.

A. The OpenSSL Repository

We added a subrepository in the openssl folder that contains a mirror of the OpenSSL git repository. From there we were able to analyze the state of OpenSSL code from several points in time. To make things easy we checked out these points in time on branches. See Table I for details.

To build this code you can read the INSTALL file. Basically the steps are as follows:

```
$ ./config shared
$ make -j
$ make install # if needed.
```

This will install ssl libraries and headers in /usr/local. If SSL is already installed, you may need to modify LD_LIBRARY_PATH to link existing executables against the compiled SSL libraries. With OpenSSL built we can do several things with it.

B. The OpenSSH Repository

One way to take advantage of the vulnerability in question was to attack SSH keys. We decided to demonstrate how such an attack would work. To do so, we used the standard ssh-keygen on the target box (one that came with the debian distribution – ssh had no built-in vulnerability other than its dependency on OpenSSL), but a heavily modified version of this program on the attacker box to iteratively generate the potential SSH keys that a machine with the flawed code could have, and try them out, using a brute-force style attack. There are two components to this: ssh-exploit-driver.bash and ssh-keyexploit.c. The ssh-exploit-driver.bash script takes an IP address and iterates over all possible PID seeds, 1-32,768, and runs ssh-keyexploit.c with each one.

See Algorithm 1 for how ssh-keyexploit.c works.

Algorithm 1 SSH key exploit

1: **procedure** EXPLOIT(IP, PID)
2:     Set the PID in the OpenSSL modified library from PID
3:     $rsa_{pub}, rsa_{priv} \leftarrow$ Modified OpenSSL generated key with PID as only source of variability.
4:     $success \leftarrow$ Try to SSH into host given by IP with rsa_{priv}.
5:     **if** $success$ **then**
6:         Report success
7:     **else**
8:         Report Failure

We run the exploit by running the script with the IP address of the target machine. With the provided virtual machines, this would look like this:

```
$ ./ssh-exploit-driver.bash 10.10.1.5
```

Eventually you get the output:

```
Success with PID seed <PID>, file in trial_identity
```

You can then SSH to the target machine with the trial_identity file.

APPENDIX B
MEASURING EFFECTS OF UNIT TESTS

As part of our analysis we wanted to see if the new unit test framework (since 2008) would catch the kind of error the debian developer introduced in 2006. Since the new random number generator, DRBG, is specification-based and much less arbitrary, this isn't an apples-to-apples comparison.

Since we can't comment out the same lines, we decided to find an area of code that's critical to updating the entropy of the DRBG random number generator. The DRBG random number generator, as specified in [12], has a few different sources of potential entropy to be mixed in to the 'pool'. The secure source of entropy, which must remain secret, is only used while seeding the DRBG. The critical activity here occurs when that entropy is hashed into the internal state of the DRBG. This occurs in drbg_hash.c in the hash_df function, which is an implementation of **Hash_df** as specified in [12]. After that, users also have the option to mix (untrusted) entropy into the random number generation process using 'additional input' strings, which are optional.

In order to see if the debian bug could happen again today with a similar error, we commented out all lines of code that updated the internal hash state of the hash-based DRBG, and ran unit tests to see if this caused them to fail. To run unit tests, complete the basic build process as described above and then type:

```
make test
```

We determined that this error is caught, see the error message in Fig.2 that was the output of make test.

Branch name	Function
`debian-mod-point`	The code that was distributed by Debian starting in 2006
`hacked-ssl-for-running-exploit`	Code we used to run the exploit.
`openssl-broken-tests-demo`	Code in version 3.0 deliberately broken to demonstrate that test coverage improved.
`master`	Current OpenSSL code

TABLE I

BRANCHES IN THE OPENSSL SUBREPOSITORY AND THEIR FUNCTION.

```
30-test_evp.t ..................... 19/?
        # INFO:  @ test/testutil/stanza.c:21
        # Reading ../../test/recipes/30-test_evp_data/evprand.txt
        # INFO:  @ test/testutil/stanza.c:122
        # Starting "CAVP Large Seed" tests at line 17
        # INFO:  @ test/testutil/stanza.c:122
        # Starting "CTR DRBG No Reseed Tests (from NIST test vectors)" tests at line 34
        # INFO:  @ test/testutil/stanza.c:122
        # Starting "Hash DRBG No Reseed Tests (from NIST test vectors)" tests at line 6324
        # ERROR: (memory) 'got == item->output' failed @ test/evp_test.c:2599
        # --- got
        # +++ item->output
        # 0000:-919f858c8a03c8a6 7eb2a0f54d2a409a 5b4bb94ecaefeefe b1c9b31ac049d127
        # 0000:+0e28130fa5ca11ed d3293ca26fdb8ae1 810611f78715082e d3841e7486f16677
        #       ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^
        # 0020:-7ccfeecc0a2a85df 16d23b14ad21b87a 5ffd238e2ae34a41 59a3fb4f39377975
        # 0020:+b28e33ffe0b93d98 ba57ba358c1343ab 2a26b4eb7940f5bc 639384641ee80a25
        #       ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^
        # 0040:-15f91a5edeca7aa1 7d30428bf31cd0dd
        # 0040:+140331076268bd1c e702ad534dda0ed8
```

Fig. 2. A test failure

APPENDIX C

MEASURING TEST COVERAGE