# 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.

```
List:       openssl-dev
Subject:    Re: Random number generator, uninit
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 act
> those 2 lines of code.  But I have no idea wh
> really has on the RNG.  The only effect I see
> might receive less entropy.  But on the other
> sure how much entropy some unitialised data h
>
Not much. If it helps with debugging, I'm in fa
(However the last time I checked, valgrind repo
error messages. Has that situation gotten bette
```

[4]

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 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

### 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, 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 ¡how many¿ machines¡CITE!¿.

Both bugs were the result of reading into memory regions that were not intended to be read into, which wouldn't even be possible in many other languages.

This raises the question of how, if possible, to catch such simple implementation errors?

## V. Investigating the current code

## 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 glossery, 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 autmated 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.

## REFERENCES


[1] https://isotoma.com/blog/2008/05/14/debians-openssl-disaste
[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.
[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
[11] https://github.com/openssl/openssl/tree/master/fuzz