WHITE PAPER

The Dangers of Using OpenSSL for Secure IoT

OpenSSL is Vulnerable

OpenSSL has made a big splash in the news recently due to the Heartbleed vulnerability, which rendered millions of Web clients and servers, as well as devices that use OpenSSL-based proprietary protocols, potentially compromisable. Potential problems include exposing private keys and user credentials—which compromises users' privacy—and leaking ASLR values—which could enable remote execution of malicious code.

The Heartbleed bug was not the first time OpenSSL was exposed as vulnerable. Indeed, there is a long history of issues: RSA PKCS #1 v1.5, incorrect checks for malformed signatures when connecting to a server whose certificates contain a DSA or ECDSA key[1], common name null termination attack[2], numerous memory/null pointer violations, timing attacks such as MAC and signing operations, various DoS attacks, RSA parasitic attacks, broken FIPS module, and other non-Heartbleed memory content leaks. The OpenSSL project itself lists a long page of known vulnerabilities: https://www.openssl.org/news/vulnerabilities.html.

The Heartbleed security bug is only the latest in a long list of OpenSSL vulnerabilities.

Many more OpenSSL security flaws could be discovered in the future. Just two months after Heartbleed, new vulnerabilities were revealed that could force weak encryption keys on public Wi-Fi users and be used to spy directly on wireless communications. Amazingly, this bug has existed in the OpenSSL code since 1998, and was never found until now[3].

Another bug discovered in June 2014 could allow remote code execution. And, in an ironic twist, some experts have noted that the efforts of system administrators to patch the Heartbleed bug have placed more systems in the range of the June security bugs than were vulnerable before[4].

Due to the wide media coverage of Heartbleed, more people have become aware of the weakness of OpenSSL and are hunting for vulnerabilities to exploit. This has led many security professionals to doubt whether the world's most used SSL/TLS implementation can be trusted in light of mass surveillance by governments and targeting by organized crime[4].

Consequently, as a result of Heartbleed and other well-publicized issues with OpenSSL, security experts, such as Prof. Alan Woodward from the Department of Computing at the University of Surrey, say that OpenSSL "is heading towards its death as a reliable form of protection"[5].

Clearly, if you use OpenSSL, your code is vulnerable. Not only is it costly to have to deal with code refactoring to address vulnerabilities as they are found, you could lose customers—before they even sign—if they decide to go with a company that doesn't use OpenSSL.

The Heartbleed security bug is only the latest in a long list of OpenSSL vulnerabilities.

Extremely Large and Complex Codebase

It's easy to assume that OpenSSL is the gold standard for SSL software, and to continue assuming that everyone has vulnerabilities, everyone has bugs, and that OpenSSL both suffers and benefits from greater scrutiny than proprietary solutions due to nature of open source software. But in reality, OpenSSL is simply too big to succeed. OpenSSL weighs in at approximately 457,000 lines of code[6]. So if we assume that OpenSSL aligns with the industry standard defect rate of 15–50[7] bugs per 1,000 lines of code, this means that there are 6855–22850 bugs in OpenSSL.

The enemy of safe, secure software is complexity. Just from a high-level directory perspective, OpenSSL exhibits a stifling level of complexity. Taking a look at the OpenSSL directory internals, we can see that the rand.h header file lives in two different directories within OpenSSL: /include/openssl and /crypto/rand.

Furthermore, the directory structure itself is very large and complex:

```
./apps
./apps/demoCA
./apps/demoCA/private
./apps/demoSRP
./apps/set
./bugs
./certs
./certs/demo
./certs/expired
./crypto
./crypto/aes
./crypto/aes/asm
./crypto/asn1
./crypto/bf
./crypto/bf/asm
./crypto/bio
./crypto/bn
./crypto/bn/asm
./crypto/bn/asm/x86
./crypto/buffer
./crypto/camellia
./crypto/camellia/asm
./crypto/cast
./crypto/cast/asm
./crypto/cmac
./crypto/cms
./crypto/comp
./crypto/conf
./crypto/des
./crypto/des/asm
     ./crypto/des/t
     ./crypto/des/times
     ./crypto/dh
     ./crypto/dsa
     ./crypto/dso
     ./crypto/ec
     ./crypto/err
     ./crypto/evp
     ./crypto/hmac
     ./crypto/idea
     ./crypto/jpake
     ./crypto/krb5
     ./crypto/lhash
     ./crypto/md2
     ./crypto/md4
     ./crypto/md5
     ./crypto/md5/asm
     ./crypto/mdc2
     ./crypto/modes
     ./crypto/modes/asm
     ./crypto/objects
     ./crypto/ocsp
     ./crypto/pem
     ./crypto/perlasm
     ./crypto/pkcs12
     ./crypto/pkcs7
     ./crypto/pkcs7/p7
     ./crypto/pkcs7/t
     ./crypto/pqueue
     ./crypto/rand
     ./crypto/rc2
     ./crypto/rc4
     ./crypto/rc4/asm
     ./crypto/rc5
     ./crypto/rc5/asm
     ./crypto/ripemd
     ./crypto/ripemd/asm
     ./crypto/rsa
     ./crypto/seed
     ./crypto/sha
     ./crypto/sha/asm
     ./crypto/srp
./crypto/stack
./crypto/store
./crypto/threads
./crypto/ts
./crypto/txt_db
./crypto/ui
./crypto/whrlpool
./crypto/whrlpool/asm
./crypto/x509
./crypto/x509v3
./demos
./demos/asn1
/demos/bio
./demos/cms
./demos/easy_tls
./demos/eay
./demos/engines
./demos/engines/cluster_
labs
./demos/engines/ibmca
./demos/engines/rsaref
./demos/engines/zencod
./demos/maurice
./demos/pkcs12
./demos/prime
./demos/sign
./demos/smime
./demos/ssl
./demos/ssltest-ecc
./demos/state_machine
./demos/tunala
./demos/x509
./doc
./doc/apps
./doc/crypto
./doc/HOWTO
./doc/ssl
./engines
./engines/ccgost
./engines/vendor_defns
./include
./include/openssl
./MacOS
./MacOS/GetHTTPS.src
./ms
./Netware
./os2
./perl
./shlib
./ssl
./test
./test/smime-certs
./times
./times/090
./times/091
./times/x86
```

When compared to simple, well-organized codebases, complex codebases that exhibit duplication are harder to maintain, harder to test, and harder to rid of bugs—all factors that increase the cost of using OpenSSL.

No Coding Guidelines

OpenSSL lacks coding guidelines, which leads to inconsistent architectural decisions, design patterns, and look and feel. These inconsistencies lead to increased complexity and decreased maintainability, which is exemplified by the following OpenSSL code snippets.

```
int BN_nist_mod_192(BIGNUM *r, const BIGNUM *a, const BIGNUM *field,
    BN_CTX *ctx)
    {
    int      top = a->top, i;
    int      carry;
    register BN_ULONG *r_d, *a_d = a->d;
    union    {
      BN_ULONG  bn[BN_NIST_192_TOP];
      unsigned int       ui[BN_NIST_192_TOP*sizeof(BN_ULONG)/sizeo-
f(unsigned int)];
      } buf;
    BN_ULONG c_d[BN_NIST_192_TOP],
      *res;
    PTR_SIZE_INT mask;
    static const BIGNUM _bignum_nist_p_192_sqr = {
      (BN_ULONG *)_nist_p_192_sqr,
      sizeof(_nist_p_192_sqr)/sizeof(_nist_p_192_sqr[0]),
      sizeof(_nist_p_192_sqr)/sizeof(_nist_p_192_sqr[0]),
      0,BN_FLG_STATIC_DATA };

    field = &_bignum_nist_p_192; /* just to make sure */

      if (BN_is_negative(a) || BN_ucmp(a,&_bignum_nist_p_192_
sqr)>=0)
      return BN_nnmod(r, a, field, ctx);

   i = BN_ucmp(field, a);
   if (i == 0)
      {
      BN_zero(r);
      return 1;
      }
   else if (i > 0)
```

```
      return (r == a) ? 1 : (BN_copy(r ,a) != NULL);

    if (r != a)
     {
      if (!bn_wexpand(r, BN_NIST_192_TOP))
         return 0;
      r_d = r->d;
      nist_cp_bn(r_d, a_d, BN_NIST_192_TOP);
      }
     else
      r_d = a_d;

      nist_cp_bn_0(buf.bn, a_d + BN_NIST_192_TOP, top - BN_
NIST_192_TOP, BN_NIST_192_TOP);

#if defined(NIST_INT64)
    {
    NIST_INT64        acc;  /*accumulator*/
    unsigned int      rp=(unsigned int *)r_d;
    const unsigned int     bp=(const unsigned int *)buf.ui;

    acc  = rp[0];    acc += bp[3*2-6];
         acc += bp[5*2-6]; rp[0] = (unsigned int)acc; acc >>= 32;

    acc += rp[1];     acc += bp[3*2-5];
         acc += bp[5*2-5]; rp[1] = (unsigned int)acc; acc >>= 32;

    acc += rp[2];    acc += bp[3*2-6];
         acc += bp[4*2-6];
         acc += bp[5*2-6]; rp[2] = (unsigned int)acc; acc >>= 32;

    acc += rp[3];    acc += bp[3*2-5];
         acc += bp[4*2-5];
         acc += bp[5*2-5]; rp[3] = (unsigned int)acc; acc >>= 32;

    acc += rp[4];    acc += bp[4*2-6];
         acc += bp[5*2-6]; rp[4] = (unsigned int)acc; acc >>= 32;

    acc += rp[5];    acc += bp[4*2-5];
         acc += bp[5*2-5]; rp[5] = (unsigned int)acc;

    carry = (int)(acc>>32);
    }
#else
   {
    BN_ULONG t_d[BN_NIST_192_TOP];
    nist_set_192(t_d, buf.bn, 0, 3, 3);
    carry = (int)bn_add_words(r_d, r_d, t_d, BN_NIST_192_TOP);
    nist_set_192(t_d, buf.bn, 4, 4, 0);
    carry += (int)bn_add_words(r_d, r_d, t_d, BN_NIST_192_TOP);
```

```
    nist_set_192(t_d, buf.bn, 5, 5, 5)
    carry += (int)bn_add_words(r_d, r_d, t_d, BN_NIST_192_TOP);
    }
#endif
    if (carry > 0)
         carry = (int)bn_sub_words(r_d,r_d,_nist_p_192[carry-1],BN_
NIST_192_TOP);
    else
         carry = 1;

/*
 * we need 'if (carry==0 || result>=modulus) result-=modulus;'
 * as comparison implies subtraction, we can write
 * 'tmp=result-modulus; if (!carry || !borrow) result=tmp;'
 * this is what happens below, but without explicit if:-) a.
 */

   mask  = 0-(PTR_SIZE_INT)bn_sub_words(c_d,r_d,_nist_p_192[0],BN_
NIST_192_TOP);     mask &= 0-(PTR_SIZE_INT)carry;
   res   = c_d;
   res   = (BN_ULONG *)
         (((PTR_SIZE_INT)res&~mask) | ((PTR_SIZE_INT)r_d&mask));
nist_cp_bn(r_d, res, BN_NIST_192_TOP);
   r->top = BN_NIST_192_TOP;
   bn_correct_top(r);

   return 1;
   }

PKCS7 *PKCS7_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509)
*certs,
        BIO *data, int flags)
{
   PKCS7 *p7;
   int i;

   if(!(p7 = PKCS7_new()))
         {
         PKCS7err(PKCS7_F_PKCS7_SIGN,ERR_R_MALLOC_FAILURE); return
NULL;
         }

   if (!PKCS7_set_type(p7, NID_pkcs7_signed))
        goto err;

   if (!PKCS7_content_new(p7, NID_pkcs7_data))
goto err;

   if (!PKCS7_content_new(p7, NID_pkcs7_data))
     goto err;
```

```
    if(!(flags & PKCS7_NOCERTS))
         {
         for(i = 0; i < sk_X509_num(certs); i++)
           {
           if (!PKCS7_add_certificate(p7, sk_X509_value(certs, i)))
               goto err;
           }
         }

     if(flags & PKCS7_DETACHED)
        PKCS7_set_detached(p7, 1);

     if (flags & (PKCS7_STREAM|PKCS7_PARTIAL))
        return p7;

     if (PKCS7_final(p7, data, flags))
        return p7;

     err:
     PKCS7_free(p7);
     return NULL;
}
```

If the OpenSSL variable names and function names were more concise and clear in their meaning, the OpenSSL codebase would be easier to maintain, easier to review, and would bear up to closer scrutiny. But as it now stands, the cost of maintaining code that integrates OpenSSL is at best unknown, and realistically the costs are significant.

Lack of Focus

OpenSSL's goal of robust and full-featured SSL/TLS stack has likely led the project astray. By attempting to be a solution for all situations, OpenSSL turns on too many options by default. For example, the dubious Heart Beat feature (RFC 6520[8]) was designed for VoIP applications to ensure continuous connectivity to a SIP server for signaling purposes. However, this feature is turned on by default for all use cases. TLS heartbeats are unnecessary, even for VoIP applications, and the desired goal could have been achieved by using TCP/IP keepalive with a BSD 4.4 select() function call.

If you take the time to research which options to turn off and how to do so, you've again increased the cost of using OpenSSL. But if you leave such unsecured and unnecessary options turned on, the potential costs associated with a security breach are infinitely higher. Either way, OpenSSL is certainly not free.

By trying to be the Jack-of-all-trades for security, OpenSSL is certainly the master of none.

OpenSSL: Under-resourced and Underfunded

Like many open source projects staffed by volunteers, OpenSSL has historically been underfunded and under-resourced. Although the discovery of Heartbleed revealed the need for more funds and personnel, OpenSSL is now adding only two permanent employees.

Advocates of open source software claim that with so many people reviewing code, errors can found more readily than in commercial software. Here we can cite Heartbleed as a notable counterexample. The Heartbleed bug was created by an expert programmer who was not expected to make such a simple coding error. If the same programmer had been working in a commercial software environment, the QA team would have been formally accountable for testing all lines of code, including his—even though he was perceived as an expert. Heartbleed highlights a phenomenon known as the "bystander effect," in which so many people are assumed to be reviewing the code that nobody actually does, and errors are passed into production[5].

Haphazard Documentation

Documentation is typically the part of a[n open source] project that gets short shrift[9]. Not only do open source projects face all the challenges to software documentation that commercial products do, such as a lack of writing skills and time and trying to keep up with ever changing software, open source projects face special challenges, such as a focus on the code development and a lack of process and tools for integrating information from email lists into formal documentation[10].

The OpenSSL documentation typifies open source documentation. On the main Documents page[11], more than half of the main topics are labeled, "[STILL INCOMPLETE]", the Wiki is, like most wikis, a jumble of bits and pieces of documentation without much order, and there's a link to "miscellaneous documents with information that has no other logical place."

It's easy to see that trying to integrate a large, complex code base such as OpenSSL with many options but a lack of cohesive and task-structured documentation is going to take a lot of time and trial and error. And the more time it takes, the more it costs to use OpenSSL.

Mocana NanoSSL is Safe

In sharp contrast to OpenSSL, Mocana NanoSSL™ has NEVER experienced a documented remote exploit attack, data bleed vulnerability, security bypass attack, signature verification or common name issue, or pointer violation error.

NanoSSL has NEVER experienced an attack.

NanoSSL is licensed as source code, which provides the review benefits of OpenSSL to white hat penetration testers, but without aiding black hat attackers.

Small and Simple Codebase

Unlike the large and complex OpenSSL codebase, NanoSSL has a simple, well organized directory structure. And the SSL client and server code portions of NanoSSL weigh in at just over 13,000 lines of code! Less code means less possibility for a defect! And unlike the big and complex OpenSSL codebase (see "Extremely Large and Complex Codebase" on page 2), the NanoSSL directory is simple:

By trying to be the Jack-of-all trades for security, OpenSSL is certainly the master of none.

```
./autotools
./bin
./docs
     ./make
     ./notes
     ./notes/v2.02
     ./notes/v2.45
     ./notes/v3.06.4
     ./notes/v3.1
     ./notes/v3.2
     ./notes/v4.0
     ./notes/v4.2
     ./notes/v5.0.1
     ./notes/v5.1
     ./notes/v5.1.1
     ./notes/v5.3
     ./notes/v5.4
./notes/v5.4.1
```

```
./notes/v5.5
./obj
./src
./src/asn1
./src/common
./src/crypto
./src/crypto/hw_offload
./src/examples
./src/harness
./src/http
./src/http/client
./src/micrium
./src/ocsp
./src/ocsp/client
./src/platform
./src/ssl
./src/ssl/client
./src/ssl/server
```

Strict Coding Standard

Unlike open source projects such as OpenSSL that cannot enforce coding standards, NanoSSL uses a strict coding standard to ensure consistent design principles and coding style, which reduces the cost of development, support, and maintenance. All NanoSSL variables and functions use descriptive names. Single-letter variable names such as 'i' are used only as counters for loops. They do not return status variables—the problem that led to the OpenSSL vulnerability relating to the ECDSA signature fail open error.

For example, OpenSSL contains the following code[12]:

```
i=ssl_verify_cert_chain(s,sk);
if ((s->verify_mode != SSL_VERIFY_NONE) && (!i)
```

This code uses non-descriptive variable names, which obfuscates its importance: verifying signatures.

In contrast, the NanoCrypto code that is the foundation of NanoSSL uses highly descriptive variables names, as seen in this method signature:

```
extern MSTATUS
DSA_verifySignature(MOC_DSA(hwAccelDescr hwAccelCtx)
const DSAKey *p_dsaDescr,
vlong *m, vlong *pR, vlong *pS,
intBoolean *isGoodSignature,
vlong **ppVlongQueue);
```

Security-Conscious, Defensive Programming

NanoSSL has built in detection for memory leaks and buffer overflows. And the numbers show that NanoSSL is well-designed and tightly written to deep data off the stack, where buffer overflows could lead to a remote execution exploit. NanoSSL allows memory to be sequestered. NanoSSL handles data appropriately by zeroizing it, religiously. NanoSSL uses less than 16KB of heap RAM during the SSL/TLS hand shake, less than 4KB for state during the open state, and keeps the stack utilization high water mark at less than 3.5KB.

NanoSSL takes a pessimistic view of the data it receives. All values are strictly checked. SSL/TLS security guards are turned on by default (for example, blinding, MAC timing resistant checks, and close on fail). Non-essential security features are off by default and turned on by compiler flags. Most importantly, NanoSSL fails closed.

The investment in security-conscious, defensive programming by Mocana ensures that you don't have to bear the cost of security breaches that result from using unsecured programs such as OpenSSL.

Professional Documentation

NanoSSL has complete, up-to-date, professional documentation, ensuring a quick, and therefore inexpensive, integration. And the user guides are backed up by an enterprise ticket-tracked support system to ensure that your questions are resolved quickly.

NanoSSL has been licensed by more than 100 device manufacturers to secure their always-on, critical Internet devices. NanoSSL has undergone tremendous scrutiny by security experts, not just casual users. NanoSSL is chosen for mission critical applications—failure is not an option.

The NanoSSL design is security-conscious; it doesn't trust anything.

Conclusions

In this white paper, we've explored some reasons why OpenSSL is vulnerable and the myriad hidden costs of using this flawed and risky code. More and more enterprises are experiencing security breaches, and the highest-level executives are being held accountable. But you can avoid such disasters by using Mocana NanoSSL —safe, proven, and easy to use security software

References

[1] https://www.openssl.org/news/secadv_20090107.txt

[2]https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf

[3]http://www.theguardian.com/technology/2014/jun/06/heartbleed-openssl-bug-security-vulnerabilities

[4]CSC, May 1, 2015 https://blogs.csc.com/2015/05/01/continued-heartburn-for-heartbleed/

[5]"Free Can Make You Bleed", April 30, 2014, SSH Communications Security, http://blog.ssh.com/free-can-make-you-bleed

[6] http://www.ohloh.net/p/openssl

[7] http://amartester.blogspot.com/2007/04/bugs-per-lines-of-code.html

[8] https://tools.ietf.org/html/rfc6520

[9] https://blog.smartbear.com/programming/14-ways-to-contribute-to-open-source-without-being-a-programming-genius-or-a-rock-star/

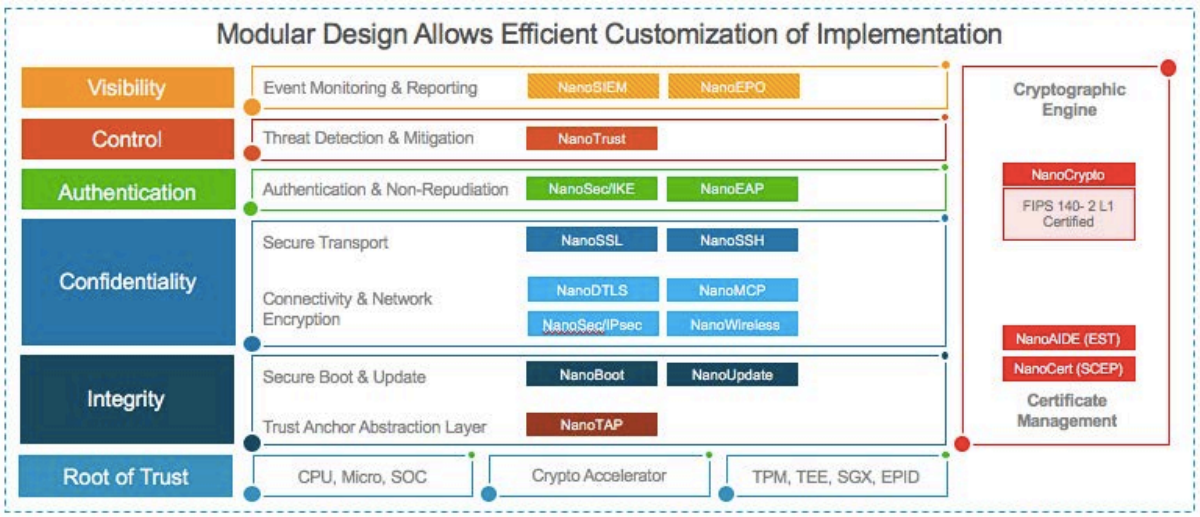
[10] http://oss-watch.ac.uk/resources/archived/documentation

[11] https://www.openssl.org/docs/

[12] http://openssl.6102.n7.nabble.com/OpenSSL-Security-Advisory-td38501.html

Mocana Security of Things™ Platform

NanoSSL is part of the Mocana Security of Things Platform™ (SoTP™), designed to secure all aspects of any connected device. As a device designer, you can choose only the components you need for your particular project or standardize company-wide on the SoTP™, future-proofing your investment with this broad, cross platform, flexible and extensible security architecture.



Protecting the Emerging Opportunity

Ultra-Secure DNA. We've been providing "MIL-spec" security technology for the most demanding, missioncritical applications for over a decade.

Scales from Device to Cloud. We provide an end-to-end, full-stack solution that enables device makers, service providers and end-users to assure security for IoT devices, services and ecosystems.

Byte-Efficient Code. Our code fits into tiny memory footprints where other implementations simply can't.

Open Source Free. Our platform contains absolutely no open source code, so you can be confident your intellectual property won't accidentally become public domain because of open source contamination.

Platform Independent. Mocana's security modules are CPU-architecture and platform independent. The SoTP™ stack supports over 35 operating systems, and 70 processors and trust anchors, today.

Easy to Use, and Re-Use. The SoTP™ modules are accessed via an extremely powerful, but simple and easy-to-use API. Our platform abstracts vendor specific requirements and makes it easier to plug your product into our software modules without extreme integration difficulties. You don't need to be a crypto or protocol expert, because we hide the complexity. And you can standardize security and re-use code across product lines and projects.

Dramatically Speeds Development Cycle and Reduces Risk. Getting security right can be hard, and getting it wrong can be costly. Mocana makes it easy to make your products secure. The SoTP™ modules are preintegrated and exhaustively tested, enabling your development team to focus on what's really important—the unique added value of your product.

About Mocana Corporation

Mocana Corporation provides mission-critical IoT security solutions for embedded systems and the Internet of Things. Founded in 2002, the company developed security software for embedded systems and mobile applications. In 2016, the company spun out the mobile application security business to focus exclusively on IoT security. Based in San Francisco, Mocana serves more than two hundred companies, including many of the largest manufacturing companies in the world that produce critical infrastructure: aerospace, chemicals, defense, electronics, energy, engineering, and transportation. We are privately held. Our investors include Shasta Ventures, Trident Capital, Sway Ventures, Southern Cross Venture Partners, GE Capital, Intel Ventures, Panasonic.

Contact US

Mocana Corporation
20 California Street, 4th floor
San Francisco, CA 94111

415-617-0055

sales@mocana.com