A Decade After Bleichenbacher '06, RSA Signature Forgery Still Works

Sze Yiu Chau - schau@purdue.edu

Black Hat USA 2019

RSA signatures, specifically the PKCS#1 v1.5 scheme, are widely used by X.509 certificates in TLS, as well as many security-critical network protocols like SSH, DNSSEC and IKE. Unfortunately, many implementations of PKCS#1 v1.5 RSA signature verification turn out to be incorrectly lenient when given malformed inputs. This white paper will explore the topic, review historic flaws and known attacks, and discuss how we applied dynamic symbolic execution to various implementations and found that many of them still suffer from different kinds of unwarranted leniency, leading to new variants of signature forgery more than a decade after the original attack was reported. I will also dissect the root causes of the flaws that we have found and give suggestions for developers to consider when facing the task of implementing similar protocols. This white paper is based on a recently published academic research paper *"Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification"* [1] co-authored by myself, Moosa Yahyazadeh (University of Iowa), Omar Chowdhury (University of Iowa), Aniket Kate (Purdue University), Ninghui Li (Purdue University).

PKCS#1 v1.5. One of the reasons why RSA signatures are so widely used, is perhaps due to its simplicity. In the 'textbook' description, given message m and public key (n, e), verifying a signature S is as simple as computing $S^e \mod n \stackrel{?}{=} H(m)$, where H is the hash function of choice. In practice, however, the output of $S^e \mod n$ contains additional information besides $H(m)$, usually following the requirements described by the PKCS#1 v1.5 signature scheme. This is because typical hash outputs (*e.g.* 160 bits for SHA-1) tend to be much shorter than the size of n (*e.g.* 2048 bits or even 4096 bits these days), and to make the scheme self-contained, the signer needs to be able to communicate the choice of H to the verifier. Hence, the PKCS#1 v1.5 signature scheme describes how padding should be done, as well as the format of the metadata used to indicate H. In short, $S^e \mod n$ is expected to look like this:

`0x00 || 0x01 || PB || 0x00 || AS`

where `PB` is the padding which has to be at least 8-byte long, made of a series of `0xFF` bytes, and `AS` is an DER-encoded ASN.1 structure, containing metadata indicating H and the actual $H(m)$.

A bit of history. Flaws in implementations of PKCS#1 v1.5 signature verification that are exploitable for signature forgery attacks were first reported by Daniel Bleichenbacher during the rump session of CRYPTO 2006 [2]. He found that some implementations are not rejecting extra trailing bytes after `AS`, and those trailing bytes can take arbitrarily any values. Because of this unwarranted leniency in the verifier, it is possible to forge signatures when e is small (*e.g.*, $e = 3$). The difficulty of achieving a successful forgery depends on the size of n and the choice of H, both of which affect the number of trailing bytes that an attacker can use. In fact, in the face of such implementation flaws, using a longer modulus (which is believed to be harder to factorize) would actually give more advantages to the attacker. The original example given by Bleichenbacher was based on a 3072 bit modulus. A follow-up analysis of the attack under shorter moduli was given by Kühn *et al.* in 2008 [3], along with variants of the attack exploiting other flaws in the verifier. For example, it was reported that, old versions of GnuTLS and OpenSSL were

not properly checking the algorithm parameter part of `AS`, which allow some bytes in the middle of `AS` to take arbitrarily any values. This can also be exploited for signature forgery if the number of bytes that are not verified is long enough. Intel Security reported in 2014 that a similar problem exists in Mozilla NSS, which can be used to forge certificates [4]. Later in 2016, Filippo Valsorda reported that the `python-rsa` implementation of PKCS#1 v1.5 signature verification does not enforce the requirement that all padding bytes need to be `0xFF` [5], which is again exploitable for signature forgery, contributing yet another attack variant to the family of Bleichenbacher '06.

New findings. The legacy of Bleichenbacher '06, however, did not end there. In our research, we revisit the problem of implementing PKCS#1 v1.5, and found that several open-source software still suffer from variants of the signature verification flaw, which can potentially be exploited for forgery attacks. The table below shows the list of software that we have investigated, and gives a summary of our findings.

Name - Version	Overly lenient	Practical exploit under small e
axTLS - 2.1.3	YES	YES
BearSSL - 0.4	No	-
BoringSSL - 3112	No	-
Dropbear SSH - 2017.75	No	-
GnuTLS - 3.5.12	No	-
LibreSSL - 2.5.4	No	-
libtomcrypt - 1.16	YES	YES
MatrixSSL - 3.9.1 (Certificate)	YES	No
MatrixSSL - 3.9.1 (CRL)	YES	No
mbedTLS - 2.4.2	YES	No
OpenSSH - 7.7	No	-
OpenSSL - 1.0.2l	No	-
Openswan - 2.6.50 *	YES	YES
PuTTY - 0.7	No	-
strongSwan - 5.6.3 *	YES	YES
wolfSSL - 3.11.0	No	-

* When using their internal implementations of PKCS#1 v1.5.

Altogether, we found that 6 software turn out to be overly lenient when it comes to PKCS#1 v1.5 signature verification. Among all the implementation flaws that we have discovered, 6 new CVEs have been assigned to the exploitable ones, 3 for axTLS, 2 for strongSwan, and 1 for Openswan.

CVE-2018-16150: We found that axTLS also accepts trailing bytes after `AS`, just like the original Bleichenbacher '06 report [2], which means the original attack will also work. In fact, our analysis found that axTLS also ignores the first 10 bytes of $S^e \mod n$, which can be exploited in tandem to make the forgery easier to succeed (reducing the number of brute force trials).

CVE-2018-16253: As shown in Snippet 1, axTLS ignores the metadata (both the algorithm object identifier and parameter) in `AS` that is used to indicate the hash algorithm, which is even laxer than the flaw of not checking algorithm parameter previously found. Hence, the forgery algorithm given in previous work [3, 4] can be adapted to apply here. This flaw happened probably because the signature verification code in axTLS is primarily used in the validation of X.509 certificates, which have a separate field for indicating the choice of signature algorithm and hash function, and one might incorrectly think that checking the metadata in `AS` is redundant.

Snippet 1: Majority of ASN.1 metadata skipped in axTLS 2.1.3

```
if (asn1_next_obj(asn1_sig, &offset, ASN1_SEQUENCE) < 0
 || asn1_skip_obj(asn1_sig, &offset, ASN1_SEQUENCE))
  goto end_get_sig;

if (asn1_sig[offset++] != ASN1_OCTET_STRING)
    goto end_get_sig;
*len = get_asn1_length(asn1_sig, &offset);
ptr = &asn1_sig[offset];          /* all ok */

end_get_sig:
    return ptr;
```

CVE-2018-16149: Additionally, we found that axTLS trusts the declared value of the length variables in `AS` without any sanity checks, which means an attacker can put absurdly large values there to trick the parser used by axTLS into performing illegal memory access. As a proof-of-concept attack, we managed to crash the signature verifier by declaring an incorrectly long $H(m)$. This attack is quite practical because axTLS performs certificate validation in a bottom-up manner, which means even if $e = 3$ is seldom used in the X.509 ecosystem these days, any MITM can potentially inject an invalid CA certificate to the chain with $e = 3$, and DoS the verifier before it gets a chance to validate the chain against the trust anchors (*e.g.*, some root CA certificates).

CVE-2018-15836: Our analysis showed that Openswan has a simple flaw of not checking the actual value of the padding bytes, similar to the problem previously discovered in `python-rsa`. Because of this, a Bleichenbacher-style low exponent signature forgery is possible.

CVE-2018-16151: We found that strongSwan does not reject signatures with extra garbage bytes hidden in the algorithm parameter portion of `AS`, a classical flaw also found previously in GnuTLS [6] and Mozilla Firefox [7].

CVE-2018-16152: Moreover, our analysis discovered that strongSwan match the algorithm object identifier in `AS` using a variant of the longest prefix match, which accepts trailing garbage bytes hidden after a valid object identifier. In other words, as long as a valid prefix is found, it does not fully consume the object identifier bytes, which means the forgery algorithm for exploiting the flaw of not properly checking algorithm parameters given in previous work [3, 4] can also be adapted to apply here.

Others: We found other peculiarities in several software but not all unwarranted leniency enables immediate practical attacks. For example, we found that MatrixSSL has two different signature verification functions, one for validating X.509 certificates, one for validating Certificate Revocation Lists (CRLs). Interestingly, the signature verification for CRL in MatrixSSL does not reject invalid algorithm object identifiers. Additionally, MatrixSSL is somewhat permissive when it comes to DER length checks, meaning that several possible length values would all be considered acceptable, and both mbedTLS and libtomcrypt share variants of this problem as well. We note that libtomcrypt also contains some other flaws that make the Bleichenbacher-style signature forgery possible, which had been found and reported independently by other researchers (CVE-2016-6129 was assigned for that).

Fixing the problems. Several fixes have been released to address the aforementioned problems. The problems in Openswan have been fixed since version 2.6.50.1 [8]. In fact, one of the forged signatures that we shared with the developers have been incorporated into the Openswan source tree as a new unit test case [9]. strongSwan has fixed the problems since version 5.7.0, and have released patches for older versions [10]. Having these problems fixed for Openswan and strongSwan is desirable as there are still key

generation programs in the IPSec ecosystem that force $e = 3$ [11]. libtomcrypt has fixed the exploitable flaw since version 1.18.0 [12]. We developed a patch for axTLS which has been incorporated into the source tree since version 2.1.5 [13].

Analysis technique. Instead of manual code review, we relied on dynamic symbolic execution (DSE) to help drive our analysis. The main intuition is that figuring out how input bytes are being consumed and verified is a problem well suited for DSE, and in the case of PKCS#1 v1.5, the several components that constitute the input buffer exhibit nice linear relations (*e.g.*, all of them together have to be as long as the size of the modulus, and the length of nodes in `AS` add up from the leaf to the root in the benign case), which can be leveraged to automatically generate meaningful concolic test cases on the fly. Additionally, we have also designed and implemented a constraint provenance tracking mechanism for `KLEE`, enabling one to identify the lines of code that contributed to the clauses of a path constraint, which makes root cause analysis much easier. For details of our analysis setup as well as the signature forgery algorithms, we refer the readers to the full paper [1].

Parsing is hard. To sum up, it was surprising that the Bleichenbacher-style low exponent signature forgery still works after more than a decade since the original report. One of the main reasons is that many software take a parsing-based approach when implementing signature verification, that is, some sort of ASN.1 parser is being used to extract $H(m)$ out of $S^e \mod n$. This is potentially problematic due to two reasons. First, given that ASN.1 is highly flexible but has a somewhat complex grammar and encoding rules, some implementations of the parser, especially if it is written primarily for PKCS#1 v1.5, try to cut corners in attempt to make the parser simpler and easier to write. Second, the robustness requirements for a generic parser could be quite different from that of a security-critical piece of code. One might reasonably expect a robust parser to not hard fail on malformed inputs and still be able to salvage useful information and let the computation proceed (see for example, [RFC 761], Sect. 2.10 Robustness Principle), but for security-critical tasks like signature verification, such leniency can often lead to exploitable flaws, and in many cases a hard failure is actually the correct behavior.

A better way of implementing PKCS#1 v1.5 signature verification, is to use a so-called construction-based approach, that is, after computing $S^e \mod n$, instead of extracting $H(m)$ out of it, the verifier can construct an "expected output", just like what the signer would do, and then compare the entire chunks of bytes. This could avoid the unenviable task of implementing a full ASN.1 parser, as signature verification now no longer depends on parsing. This is similar to what had been done before for fixing the problems found several years ago [14], and is how we fixed the newly found problems in axTLS [15].

In the long run, perhaps it is worth reconsidering the design of incorporating a flexible but complex structure inside security-critical objects like digital signatures. While an ASN.1 DER structure like `AS` is highly extensible and can easily accommodate new hash algorithms, the reality is, new standardized algorithms seldom get introduced, and complicating a common but critical routine that gets invoked multiple times daily for a flexibility that is enjoyed only once in a while might not seem to be worthwhile.

References

[1] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, "Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification," in *The Network and Distributed System Security Symposium (NDSS) 2019*.

[2] H. Finney, *Bleichenbacher's RSA signature forgery based on implementation error*, 2006 (accessed Jul 14, 2019), https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html.

[3] U. Kühn, A. Pyshkin, E. Tews, and R. Weinmann, "Variants of bleichenbacher's low-exponent attack on PKCS#1 RSA signatures," in *Sicherheit 2008: Sicherheit, Schutz und Zuverlässigkeit. Konferenzband der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 2.-4. April 2008 im Saarbrücker Schloss.*, 2008, pp. 97–109.

[4] Intel Security: Advanced Threat Research, *BERserk Vulnerability – Part 2: Certificate Forgery in Mozilla NSS*, 2014 (accessed Jul 14, 2019), https://bugzilla.mozilla.org/attachment.cgi?id=8499825.

[5] F. Valsorda, *Bleichenbacher'06 signature forgery in python-rsa*, 2016 (accessed Jul 14, 2019), https://blog.filippo.io/bleichenbacher-06-signature-forgery-in-python-rsa/.

[6] S. Josefsson, *[gnutls-dev] Original analysis of signature forgery problem*, 2006 (accessed Jul 21, 2018), https://lists.gnupg.org/pipermail/gnutls-dev/2006-September/001240.html.

[7] Bugzilla, *RSA PKCS#1 signature verification forgery is possible due to too-permissive SignatureAlgorithm parameter parsing*, 2014 (accessed Jul 18, 2018), https://bugzilla.mozilla.org/show_bug.cgi?id=1064636.

[8] *[Openswan Users] Xelerance has released Openswan 2.6.50.1*, 2018 (accessed Jul 16, 2019), https://lists.openswan.org/pipermail/users/2018-August/023761.html.

[9] *wo#7449 . test case for Bleichenbacher-style signature forgery*, 2018 (accessed Jul 16, 2019), https://github.com/xelerance/Openswan/commit/937d24f88566702d72a549e9e8650320cb4f95cf.

[10] *strongSwan Vulnerability (CVE-2018-16151, CVE-2018-16152)*, 2018 (accessed Jul 16, 2019), https://www.strongswan.org/blog/2018/09/24/strongswan-vulnerability-(cve-2018-16151,-cve-2018-16152).html.

[11] *Ubuntu Manpage: ipsec_rsasigkey - generate RSA signature key*, 2018 (accessed Jul 16, 2019), http://manpages.ubuntu.com/manpages/bionic/man8/ipsec_rsasigkey.8.html.

[12] *libtomcrypt 1.18.0 Release Note*, 2017 (accessed Jul 16, 2019), https://www.libtom.net/news/LTC_1.18.0/.

[13] *[axtls-general] v2.1.5 of axTLS released*, 2019 (accessed Jul 16, 2019), https://sourceforge.net/p/axtls/mailman/message/36613862/.

[14] *PKCS#1 signature validation*, 2014 (accessed Jul 16, 2019), https://www.imperialviolet.org/2014/09/26/pkcs1.html.

[15] *Apply CVE fixes for X509 parsing*, 2018 (accessed Jul 16, 2019), https://github.com/igrr/axtls-8266/commit/5efe2947ab45e81d84b5f707c51d1c64be52f36c.