The real impact of obsolete cryptography, applied to SSL/TLS

Olivier Levillain

ANSSI

2016-07-06

Foreword

http://paperstreet.picty.org/cyberinbretagne

olivier.levillain@ssi.gouv.fr

Who am I?

Olivier Levillain (`@pictyeye`)

- M2 internship in cryptography: study of a hash function
- member of the systems lab at ANSSI (2007-2012)
- head of the network lab at ANSSI (2012-2015)
- head of the training center (CFSSI) (2015-)

Research

- part of the low-level x86 security work (SMM/ACPI)
- PhD student working on SSL/TLS (defense in September)
- Participation to languages studies since 2007
- some work on binary *parsers*

Teaching

- cryptography: hash function and cryptanalysis
- systems module for the CFSSI
- courses on SSL/TLS, and more recently on secure development

ANSSI

ANSSI (French Network and Information Security Agency) has InfoSec (and no Intelligence) missions:

- detect and early react to cyber attacks
- prevent threats by supporting the development of trusted products and services
- provide reliable advice and support
- communicate on information security threats and the related means of protection

These missions concern:

- governmental entities
- companies
- the general public

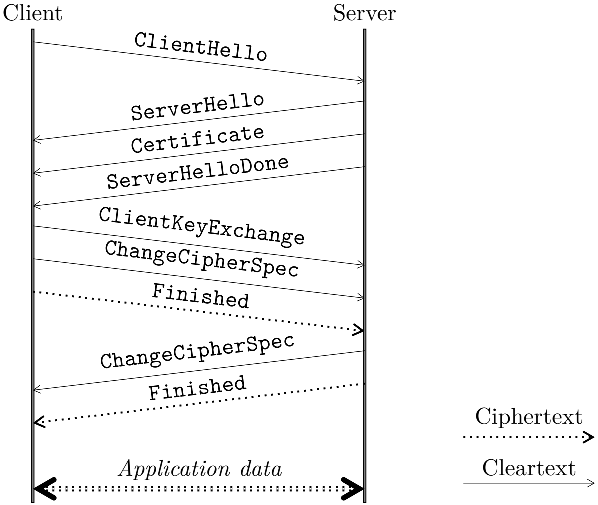
SSL/TLS: an essential building block of Internet

- `https://` invented by Netscape in 1995
  - the beginning of the e-commerce
- Massive usage of SSL/TLS today
  - HTTPS, well beyond e-commerce websites
  - A way to secure other protocols (SMTP, IMAP, LDAP...)
  - SSL VPN
  - EAP TLS

SSL/TLS: an essential building block of Internet

- `https://` invented by Netscape in 1995
  - the beginning of the e-commerce
- Massive usage of SSL/TLS today
  - HTTPS, well beyond e-commerce websites
  - A way to secure other protocols (SMTP, IMAP, LDAP...)
  - SSL VPN
  - EAP TLS
- SSL (*Secure Sockets Layer*) or TLS (*Transport Layer Security*) ?
  - SSLv2 (1995) and v3 (1996) designed by Netscape
  - TLS 1.0 (2001) a.k.a. SSLv3.1, handled by IETF
  - New revisions since: 1.1 (2006), 1.2 (2008) and 1.3 (2016?)

SSL/TLS in a nutshell



SSL/TLS goals and their solutions

SSL/TLS bottom line is:

- an authenticated key exchange
- a secure channel providing confidentiality and integrity protection

SSL/TLS goals and their solutions

SSL/TLS bottom line is:

- an authenticated key exchange
- a secure channel providing confidentiality and integrity protection

Known solutions to these problems

- SIGMA (Sign-and-MAC) (Krawczyck, 2003)
- AEAD schemes, such as OCB (Rogaway, 2001), CCM (Housley et al., 2003), GCM (McGrew et al., 2005).. or simply CTR-then-HMAC

SSL/TLS goals and their solutions

SSL/TLS bottom line is:

- an authenticated key exchange
- a secure channel providing confidentiality and integrity protection

Known solutions to these problems

- SIGMA (Sign-and-MAC) (Krawczyck, 2003)
- AEAD schemes, such as OCB (Rogaway, 2001), CCM (Housley et al., 2003), GCM (McGrew et al., 2005).. or simply CTR-then-HMAC

TLS 1.2 seems to include (some form of) these solutions. Yet,

- it is always possible to negotiate something else
- the devil is in the details...

SSL/TLS goals and the reality (1/2)

Authentication and key exchange

- RSA encryption (with implicit authentication of the server)
- Signed ephemeral Diffie-Hellman (finite fields or elliptic curves)

SSL/TLS goals and the reality (1/2)

Authentication and key exchange

- RSA encryption (with implicit authentication of the server)
- Signed ephemeral Diffie-Hellman (finite fields or elliptic curves)
- *Fixed Diffie-Hellman*, *Kerberos*, *PSK and SRP*

SSL/TLS goals and the reality (1/2)

Authentication and key exchange

- RSA encryption (with implicit authentication of the server)
- Signed ephemeral Diffie-Hellman (finite fields or elliptic curves)
- *Fixed Diffie-Hellman*, *Kerberos*, *PSK and SRP*

Problems with Diffie-Hellman groups (mostly finite fields)

- chosen by the server
- no negotiation: the client must accept or abort the connection

SSL/TLS goals and the reality (1/2)

Authentication and key exchange

- RSA encryption (with implicit authentication of the server)
- Signed ephemeral Diffie-Hellman (finite fields or elliptic curves)
- *Fixed Diffie-Hellman*, *Kerberos*, *PSK and SRP*

Problems with Diffie-Hellman groups (mostly finite fields)

- chosen by the server
- no negotiation: the client must accept or abort the connection
- some servers propose 256- or 512-bit FFDH groups
- some clients choke on 2048-bit FFDH groups
- most connections end up using a shared 1024-bit group

SSL/TLS goals and the reality (2/2)

Symmetric encryption and integrity protection

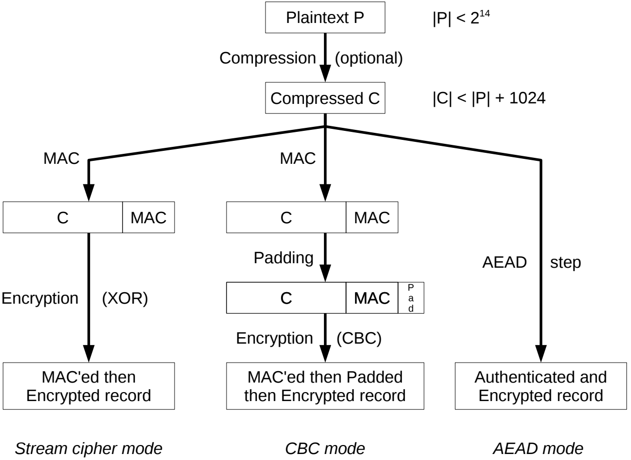
- HMAC-then-RC4
- HMAC-then-CBC

SSL/TLS goals and the reality (2/2)

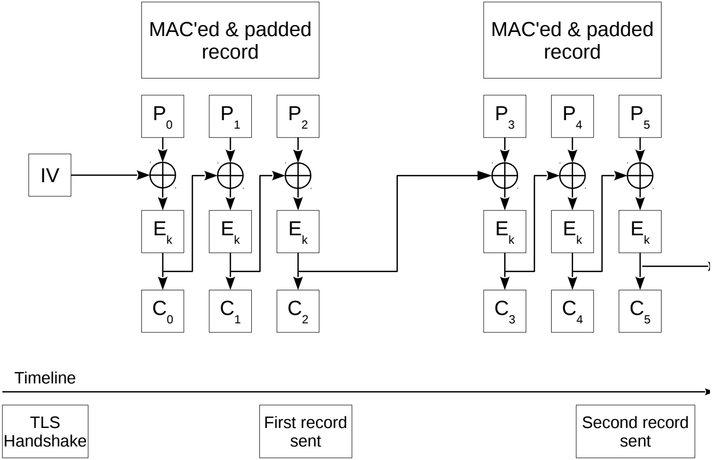
Symmetric encryption and integrity protection

- HMAC-then-RC4
- HMAC-then-CBC
- GCM or CCM (only with TLS 1.2)
- CBC-then-HMAC (only with a recent and undeployed extension)

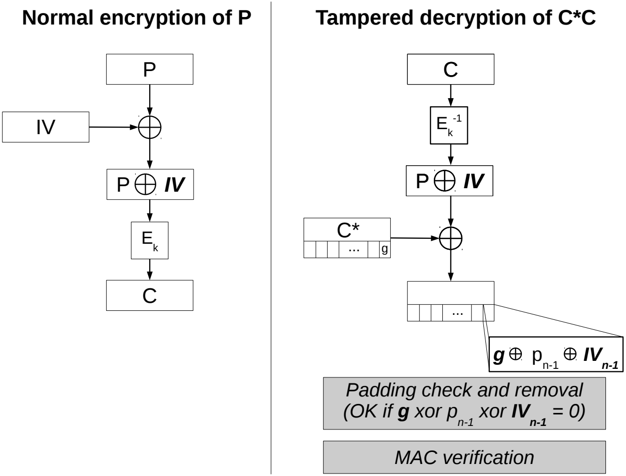
The Record Protocol



Mac-then-CBC in TLS



Mac-then-CBC: the issue



Mac-then-CBC: a practical example

Demo, using Python

Mac-then-CBC: padding oracle in TLS?

What about TLS?

- there is an implementation note in TLS 1.1 to help counter timing-based padding oracles

  > *In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet.*

- as soon as a cryptographical error arises, the connection is shut down and the traffic keys are erased

So, these attacks are not applicable to TLS?

Lucky 13

Let's read the rest of the implementation note:

> *This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.*

Lucky 13

Let's read the rest of the implementation note:

> *This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.*

What about the connection shutdown problem?

- it does not exist in DTLS (Paterson et al., 2013)
- it does not matter when the attacker targets a secret repeated across different connections (Lucky 13, AlFardan et al., 2014)

POODLE

With SSLv3, the padding is not completely specified: only the last byte of the block must contain the padding length (minus one)

POODLE vs Lucky 13

- all SSLv3 implementations are reliable padding oracle *by design*
- since the attacker only learns something about the last byte, the plaintext must be malleable to align the blocks accordingly

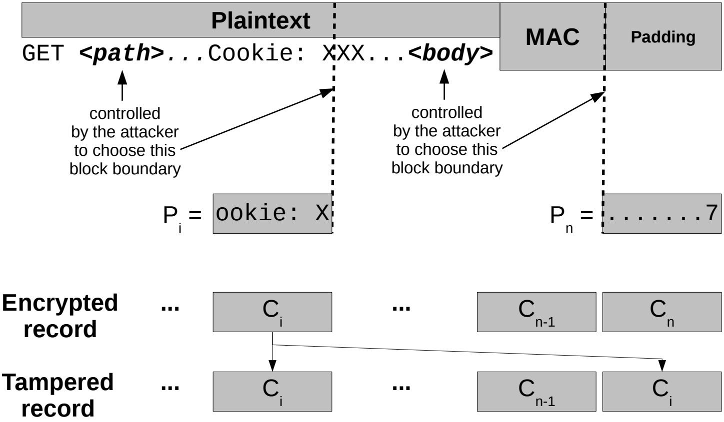
POODLE

With SSLv3, the padding is not completely specified: only the last byte of the block must contain the padding length (minus one)

POODLE vs Lucky 13

- all SSLv3 implementations are reliable padding oracle *by design*
- since the attacker only learns something about the last byte, the plaintext must be malleable to align the blocks accordingly
- bonus (for the attacker): some TLS implementations were vulnerable to POODLE

POODLE attack within HTTPS



Interesting thoughts regarding CBC mode

`https://www.openssl.org/~bodo/tls-cbc.txt`

Bodo Möller

Interesting thoughts regarding CBC mode

`https://www.openssl.org/~bodo/tls-cbc.txt`

Bodo Möller

- Beware of padding oracles with CBC mode in TLS
- CBC mode with implicit IV may be subject to attacks
- With SSLv3, the padding is not well specified, which aggravates padding oracles

Interesting thoughts regarding CBC mode

`https://www.openssl.org/~bodo/tls-cbc.txt`

Bodo Möller (2004-05-20)

- Beware of padding oracles with CBC mode in TLS
- CBC mode with implicit IV may be subject to attacks
- With SSLv3, the padding is not well specified, which aggravates padding oracles
- ... he was describing Lucky 13, BEAST and POODLE

Solutions?

Different solutions were proposed:

Solutions?

Different solutions were proposed:

- prefer RC4

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*
  - even knowing that, you can still get it wrong (Lucky microseconds, an attack against s2n, Paterson et al., 2015)

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*
  - even knowing that, you can still get it wrong (Lucky microseconds, an attack against s2n, Paterson et al., 2015)
- use Encrypt-then-MAC (RFC 7366)

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*
  - even knowing that, you can still get it wrong (Lucky microseconds, an attack against s2n, Paterson et al., 2015)
- use Encrypt-then-MAC (RFC 7366)
  - not really implemented in practice

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*
  - even knowing that, you can still get it wrong (Lucky microseconds, an attack against s2n, Paterson et al., 2015)
- use Encrypt-then-MAC (RFC 7366)
  - not really implemented in practice
- use AEAD

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*
  - even knowing that, you can still get it wrong (Lucky microseconds, an attack against s2n, Paterson et al., 2015)
- use Encrypt-then-MAC (RFC 7366)
  - not really implemented in practice
- use AEAD
  - this requires TLS 1.2

Solutions?

Different solutions were proposed:

- prefer RC4
  - but RC4 has exploitable probabilistic biases (Paterson et al., 2014)
- implement MAC-then-CBC carefully
  - following the implementation note from TLS 1.1 is insufficient
  - writing a real constant time implementation is *hard*
  - even knowing that, you can still get it wrong (Lucky microseconds, an attack against s2n, Paterson et al., 2015)
- use Encrypt-then-MAC (RFC 7366)
  - not really implemented in practice
- use AEAD
  - this requires TLS 1.2
  - implementation errors are still possible (nonce reuse breaks authenticity in GCM, Böck et al., 2016)

The future?

TLS 1.3:

- AEAD is the only way left in record protocol!
- nonce definition should be cleaned up
- bonus: no more compression (CRIME and related attacks)
- bonus: variable length padding to hide record length

The future?

TLS 1.3:

- AEAD is the only way left in record protocol!
- nonce definition should be cleaned up
- bonus: no more compression (CRIME and related attacks)
- bonus: variable length padding to hide record length

Even with robust TLS 1.3 implementations

- deployment issues
- legacy versions persistence

Lessons learned/still to learn

- CBC oracle have been known for a long time
- TLS 1.1 and TLS 1.2 were published without taking care of the problem
- SSLv3 (1995) was still widely supported in 2014!

Lessons learned/still to learn

- CBC oracle have been known for a long time
- TLS 1.1 and TLS 1.2 were published without taking care of the problem
- SSLv3 (1995) was still widely supported in 2014!
- Lesson 1: there is a crucial need to propose, deploy, then force the use of new cryptographical constructions, as soon as possible

A fundamental issue with RSA encryption

In the initial diagram, we showed an RSA encryption key exchange:

- the server presents its RSA certificate (and the corresponding chain)
- the client chooses a random pre-master secret PMS
- the client encrypts PMS with the server RSA public key
- both the client and the server know the PMS and derive the traffic keys from it
- the server is implictly authenticated

A fundamental issue with RSA encryption

In the initial diagram, we showed an RSA encryption key exchange:

- the server presents its RSA certificate (and the corresponding chain)
- the client chooses a random pre-master secret PMS
- the client encrypts PMS with the server RSA public key
- both the client and the server know the PMS and derive the traffic keys from it
- the server is implictly authenticated

However, as soon as the long term secret (the RSA private key) is compromised, so are all the previous sessions.

- we do not have the *forward secrecy* property

Details of PKCS#1 v1.5 padding

PKCS#1 v1.5 encryption format:

$\longleftarrow$ n bytes $\longrightarrow$

`00`	`02`	`rr`	...	`rr`	`00`	`Data to encrypt`

$\leftarrow$ 8+ bytes $\rightarrow$

Principle of the Million Message Attack

Bleichenbacher devised an attack in 1998 against this padding scheme

- every correctly padded plaintext starts with 00 02
- the plaintext must correspond to a big integer between 2^{n-16} and 3^{n-16} (with an n-bit modulus)

Principle of the Million Message Attack

Bleichenbacher devised an attack in 1998 against this padding scheme

- every correctly padded plaintext starts with 00 02
- the plaintext must correspond to a big integer between 2^{n-16} and 3^{n-16} (with an n-bit modulus)
- some RSA implementations allow for padding oracle:
  - the simplest form consists in advertising the presented condition
  - finer-grain conditions can also be advertised
- an attacker can use this to recover the P from a ciphertext C
- she asks for the decryption of $C \cdot X^e$ with a known X value
- each valid response gives the attacker one more equation

Principle of the Million Message Attack

Bleichenbacher devised an attack in 1998 against this padding scheme

- every correctly padded plaintext starts with 00 02
- the plaintext must correspond to a big integer between 2^{n-16} and 3^{n-16} (with an n-bit modulus)
- some RSA implementations allow for padding oracle:
  - the simplest form consists in advertising the presented condition
  - finer-grain conditions can also be advertised
- an attacker can use this to recover the P from a ciphertext C
- she asks for the decryption of $C \cdot X^e$ with a known X value
- each valid response gives the attacker one more equation
- initially, the attack required around one million messages to recover a given plaintext for a 1024-bit RSA
- several optimisations were then presented (e.g. Bardou et al., 2012)

Application to TLS

The RSA decryption takes place during the Handshake phase

- all the messages are exchanged in cleartext
- the client is never authenticated

Application to TLS

The RSA decryption takes place during the Handshake phase

- all the messages are exchanged in cleartext
- the client is never authenticated

Originally, SSL specs did not tell how to handle ill-padded messages

- a common behavior was to offer some form of a padding oracle

A useful implementation note

TLS 1.0 proposed an efficient countermeasure

1. the server draws a random fake pre-master secret
2. it tries to decrypt the RSA message
   - 2.1 if all goes well, it returns the corresponding plaintext (the actual pre-master secret sent by the client)
   - 2.2 if an error arises, it returns the fake pre-master secret
3. it continues the handshake in both cases

A useful implementation note

TLS 1.0 proposed an efficient countermeasure

1. the server draws a random fake pre-master secret
2. it tries to decrypt the RSA message
   - 2.1 if all goes well, it returns the corresponding plaintext (the actual pre-master secret sent by the client)
   - 2.2 if an error arises, it returns the fake pre-master secret
3. it continues the handshake in both cases

Notes:

- in the fake case, the client can not distinguish the server-sent messages from valid ones
- to avoid timing leaks, the fake value must be generated *first*

A more recent oracle in Java

In 2014, Meyer et al. found the Java TLS implementation to be vulnerable to Bleichenbacher attack

A more recent oracle in Java

In 2014, Meyer et al. found the Java TLS implementation to be vulnerable to Bleichenbacher attack

The if part of the plan was indeed handled using an exception handler... which takes time to execute and can be observed!

Another recent flaw: DROWN

In 2016, Aviram et al. presented DROWN, an attack using an SSLv2 server as a padding oracle

Another recent flaw: DROWN

In 2016, Aviram et al. presented DROWN, an attack using an SSLv2 server as a padding oracle

- the target is still an RSA-encrypted pre-master secret (the `ClientKeyExchange` message from a TLS session)

Another recent flaw: DROWN

In 2016, Aviram et al. presented DROWN, an attack using an SSLv2 server as a padding oracle

- the target is still an RSA-encrypted pre-master secret (the `ClientKeyExchange` message from a TLS session)
- the oracle is provided by an SSLv2 server using *the same RSA key* as the target server
  - the same HTTPS server
  - or a mail server for the related domain

Another recent flaw: DROWN

In 2016, Aviram et al. presented DROWN, an attack using an SSLv2 server as a padding oracle

- the target is still an RSA-encrypted pre-master secret (the `ClientKeyExchange` message from a TLS session)
- the oracle is provided by an SSLv2 server using *the same RSA key* as the target server
  - the same HTTPS server
  - or a mail server for the related domain
- forcing the use of EXPORT ciphersuite, testing the validity of an RSA message requires 2^{40} computations

Another recent flaw: DROWN

In 2016, Aviram et al. presented DROWN, an attack using an SSLv2 server as a padding oracle

- the target is still an RSA-encrypted pre-master secret (the `ClientKeyExchange` message from a TLS session)
- the oracle is provided by an SSLv2 server using *the same RSA key* as the target server
  - the same HTTPS server
  - or a mail server for the related domain
- forcing the use of EXPORT ciphersuite, testing the validity of an RSA message requires 2^{40} computations

Bonuses

- SSLv2 and EXPORT ciphersuites are still widely supported
- A bug in OpenSSL made them impossible to disable in SSLv2
- Another bug in the SSLv2 let to a more efficient oracle

Lessons learned/still to learn

- RSA PKCS# v1.5 encryption: an obsolete and dangerous scheme
- Lesson 1: use up-to-date crypto, and reject obsolete constructions

Lessons learned/still to learn

- RSA PKCS# v1.5 encryption: an obsolete and dangerous scheme
- Lesson 1: use up-to-date crypto, and reject obsolete constructions
- Lesson 2: cryptographical material separation should be enforced when possible, to allow for damage control/defense-in-depth

PKCS#1 v1.5 RSA signature scheme

- m, the message to sign
- (N, d), the RSA private key
- H, a hash function

PKCS#1 v1.5 RSA signature scheme

- m, the message to sign
- (N, d), the RSA private key
- H, a hash function

- Hash: compute $h = H(m)$;
- Format: prepare an ASN.1 DER block, encoding h, the digest info d
- Pad: prepend d with `00 01 ff ... ff 00`
- Sign: let x be the big integer represented by this value, the final result is $s = x^d[N]$.

PKCS#1 v1.5 RSA signature scheme

- m, the message to sign
- (N, d), the RSA private key
- H, a hash function

- Hash: compute $h = H(m)$;
- Format: prepare an ASN.1 DER block, encoding h, the digest info d
- Pad: prepend d with `00 01 ff ... ff 00`
- Sign: let x be the big integer represented by this value, the final result is $s = x^d[N]$.

Usage

- standard RSA signatures in X.509 certificates
- TLS RSA signature for DHE-RSA ciphersuites

More ASN.1 grammar

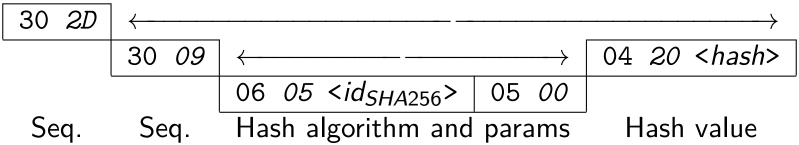
PKCS#1 v1.5 signature format:

← *n* bytes →

00	01	ff	...	ff	00	DigestInfo

← 8+ bytes →

DigestInfo:



The original Bleichenbacher attack on RSA signature

Consider the following implementation to verify an RSA signature

- exponentiate s to the e-th power
- remove the padding from the resulting string
- parse an ASN.1 object from the unpadded blob *without checking that bytes are remaining at the end of the blob*

The original Bleichenbacher attack on RSA signature

Consider the following implementation to verify an RSA signature

- exponentiate s to the e-th power
- remove the padding from the resulting string
- parse an ASN.1 object from the unpadded blob *without checking that bytes are remaining at the end of the blob*

The following plaintext would be accepted:

$\longleftarrow$ n bytes $\longrightarrow$

`00`	`01`	`ff`	...	`ff`	`00`	`DigestInfo`	*garbage*

$\leftarrow p$ bytes $\rightarrow$

Bleichenbacher vs RSA signature

Demo: forging an RSA signature for an arbitrary digest info

Hypotheses:

- the public key used to verify the signature uses $e = 3$
- the code verifying the signature ignores trailing bytes after the ASN.1

Impact on TLS

It is possible to forge an arbitrary signature for a public key with $e = 3$ that will be accepted by a vulnerable implementation

Impact on TLS

It is possible to forge an arbitrary signature for a public key with $e = 3$ that will be accepted by a vulnerable implementation

If the public key is the server key

- the attacker can forge the signature for the `ServerKeyExchange`
- she can impersonate the server

Impact on TLS

It is possible to forge an arbitrary signature for a public key with $e = 3$ that will be accepted by a vulnerable implementation

If the public key is the server key

- the attacker can forge the signature for the `ServerKeyExchange`
- she can impersonate the server

If the public key belongs to a trusted certificate authority (CA)

- the attacker can forge the signature for any certificate
- unless the CA is constrained otherwise...
- this means the attacker can impersonate any server!

Concrete attack in 2014

A variant was published in September 2014:

- the attack also requires a public exponent equals to 3
- it relies on a liberal parser, accepting alternate length representations
  - 32 must be represented as `0x20` in DER
  - in BER, it can be represented as `0x8f` followed by 14 zeroes and `0x20`
- the last hypothesis is that there is an integer overflow: the zeroes do not really need to be null!

Concrete attack in 2014

A variant was published in September 2014:

- the attack also requires a public exponent equals to 3
- it relies on a liberal parser, accepting alternate length representations
  - 32 must be represented as `0x20` in DER
  - in BER, it can be represented as `0x8f` followed by 14 zeroes and `0x20`
- the last hypothesis is that there is an integer overflow: the zeroes do not really need to be null!

Impact

- several vulnerable implementations, including NSS
- there exists unconstrained trusted CA with $e = 3$
- Mozilla products could be universally fooled!

Concrete attack in 2014

A variant was published in September 2014:

- the attack also requires a public exponent equals to 3
- it relies on a liberal parser, accepting alternate length representations
  - 32 must be represented as `0x20` in DER
  - in BER, it can be represented as `0x8f` followed by 14 zeroes and `0x20`
- the last hypothesis is that there is an integer overflow: the zeroes do not really need to be null!

Impact

- several vulnerable implementations, including NSS
- there exists unconstrained trusted CA with $e = 3$
- Mozilla products could be universally fooled!
- by the way, the same day, Shellshock (a bash flaw) was published

How to fix NSS universal forgery?

The obvious fix: detect the integer overflow

How to fix NSS universal forgery?

The obvious fix: detect the integer overflow

The other obvious fix: forbid non canonical BER representations

How to fix NSS universal forgery?

The obvious fix: detect the integer overflow

The other obvious fix: forbid non canonical BER representations

A good idea: stop using $e = 3$ (and 5, 7...)

How to fix NSS universal forgery?

The obvious fix: detect the integer overflow

The other obvious fix: forbid non canonical BER representations

A good idea: stop using $e = 3$ (and 5, 7...)

The elegant and reliable solution:

- instead of parsing the ASN.1 to compare the hashes,
- build the expected ASN.1 from the expected hash and compare the corresponding digest infos

How to fix NSS universal forgery?

The obvious fix: detect the integer overflow

The other obvious fix: forbid non canonical BER representations

A good idea: stop using $e = 3$ (and 5, 7...)

The elegant and reliable solution:

- instead of parsing the ASN.1 to compare the hashes,
- build the expected ASN.1 from the expected hash and compare the corresponding digest infos
- *but it breaks with non compliant CAs forgetting the null params...*

Lessons learned/still to learn

- RSA PKCS# v1.5 encryption: an obsolete and dangerous scheme
- Lesson 1: use up-to-date crypto, and reject obsolete constructions
- Lesson 2: use defense-in-depth mechanisms (here: avoid $e = 3$)

Lessons learned/still to learn

- RSA PKCS# v1.5 encryption: an obsolete and dangerous scheme
- Lesson 1: use up-to-date crypto, and reject obsolete constructions
- Lesson 2: use defense-in-depth mechanisms (here: avoid $e = 3$)
- Lesson 3: implementating cryptography (and using ASN.1) is tricky. Keep the specs simple!

Lessons

- Use up-to-date crypto, and reject obsolete constructions
  - how to handle the transition smoothly and quickly
- Use defense-in-depth mechanisms
  - sometimes this has a performance cost
- Keep the specs simple and audit your implems
  - this can be long and hard, but TLS 1.3 is coming!

TLS 1.3: a new hope?

TLS 1.3: a major cleanup

- only up-to-date constructions are allowed
- anti-downgrade mechanisms are being proposed
- the 1-RTT mode is clean and simple
- a lot of work has been done to prove TLS 1.3

TLS 1.3: a new hope?

TLS 1.3: a major cleanup

- only up-to-date constructions are allowed
- anti-downgrade mechanisms are being proposed
- the 1-RTT mode is clean and simple
- a lot of work has been done to prove TLS 1.3

But...

- TLS 1.2 might still be here for a long time
- the 0-RTT is complex and is still under discussion at the IETF WG
- what about the quality of common implementations?

Questions

Thank you for your attention

`olivier.levillain@ssi.gouv.fr`