Hash Collisions break TLS Signatures

1. INTRODUCTION

Since the discovery of collisions in MD5 and near-collisions in SHA-1, the use of these hash algorithms in various protocols have come under question [4]. We consider the security of client and server signatures in the TLS protocol. In particular, does authentication in TLS rely on a collision resistant hash function or would a second preimage-resistant hash function be sufficient?

It is commonly believed that current collision attacks on MD5 affect only non-repudiable signatures such as those on documents and PKI certificates, but not to the signatures used within protocols, since these signatures are over structured text and include nonces [3].[1]

On the contrary, we show that if TLS signatures use a hash function for which chosen-prefix collisions can be computed in real-time, then a man-in-the-middle attacker can impersonate TLS 1.2 [1] clients and TLS 1.3 [2] servers. Moreover, if the server nonce is chosen by the client (hypothetically), then TLS 1.2 server signatures can be impersonated. Finally, if the client nonce is fresh but predictable, then all of these attacks can be converted to practical offline attacks.

2. COLLIDING TLS 1.2 CLIENT AUTHENTICATION

- Suppose a server S authenticates a client C using TLS client authentication
- Suppose the client C is willing to also use the same certificate at another server M
- Suppose C is willing to create signatures with a hash function for which chosen-prefix collisions are easy to compute (e.g. RSA-MD5)
- Then, M can force C to create a signature that M can use to impersonate C at S

In a client-authenticated TLS 1.2 (EC)DHE handshake, C hashes and signs the following transcript

`CH | SH | SC | SKE | SCR | SHD | CC | CKE`

- `ClientHello`(CH): has a fresh random value (`cr`), and an arbitrary number of extensions; we denote the last extension `ext`.
- `ServerHello`(SH): has a fresh random value (`sr`)
- `ServerCertificate`(SC): has the server's certificate (`scert`)
- `ServerKeyExchange`(SKE): has the server's (EC)DHE key share and signature
- `CertificateRequest`(SCR): has a certificate request with an arbitrary number of certificate authorities; we denote the last authority `auth`.
- `ServerHelloDone`(SHD): completes the server flight
- `ClientCertificate`(CC): has the client's certificate (`ccert`)
- `ClientKeyExchange`(CKE): has the client's (EC)DHE key share

Suppose C connects to the malicious server M and sends `CH(cr)`. M will respond to C with a seqence of messages `SH' SC' SKE' SCR'(auth') SHD'`, where we show how `auth`' is computed below. Then C will send its own certificate `CC` and key exchange message `CKE` and then sign the full transcript: `CH SH' SC' SKE' SCR'(auth') SHD' CC CKE`.

Independently, M connects to the honest server S and sends `CH'(cr',ext')`, where we show how `ext`' is computed below. Suppose S responds with a sequence of messages `SH SC SKE SCR SHD`. To impersonate C, M now forwards C's certificate and key exchange messages `CC CKE` and has to produce a signature over the transcript:
`CH'(cr',ext') SH SC SKE SCR SHD CC CKE`.

Let us suppose the size of the server S's message sequence `SH SC SKE SCR` is known in advance to be L. To compute `ext`' and `auth`', M finds a hash collision with the following two chosen prefixes:

```
P1 = CH(cr) SH' SC' SKE' SCR'(auth0)
P2 = CH'(cr',ext0)
```

[1] `http://www.rtfm.com/dimacs.pdf`,`http://events.iaik.tugraz.at/HashWorkshop07/slides/ekr_Indigestion.pdf`

In P1, the SCR' message has an empty last certificate authority auth0 whose length is set to C + L, where C is the number of collision bytes desired. That is, P1 still needs C+L bytes to be added to complete the SCR' message. In P2, the CH' message has an empty last extension ext0 whose length is set to C. That is, P2 still needs C bytes to be added to complete the CH' message.

Now, M computes two byte array B1 and B2 of C bytes that make Hash(P1 B1) = Hash(P2 B2). Note that auth' still has room to accommodate the L bytes of S's response SH SC SKE SCR. We then add a common suffix as follows:

```
T1 = P1 B1 SH SC SKE CR SHD CC CKE
T2 = P2 B2 SH SC SKE CR SHD CC CKE
```

Here, T1 corresponds to the transcript that C is willing to sign for M, and T2 corresponds to the transscript that M needs to sign for S. Effectively, we have set ext' to ext0 B2 and we have set auth' to auth0 SH SC SKE SCR. Now Hash(T1) = Hash(T2) even though the two traces have different certificates and different key exchanges.

Hence, M can get C to sign T1 and then forward the signature to S pretending to be C.

Exploiting the collision.

The collision on C's digital signature already violates the high-level client authentication goal, but we need one more step to ensure that M can complete the handshake with S and fully impersonate C. M needs to know the master secret established in the handshake with transcript T2.

Suppose C and M are engaged in a DHE handshake. M forces C to choose a public value for which it already knows the secret exponent as follows. In its SKE' message to C, M sends a bogus Diffie-Hellman group (p,g) with a non-prime p = k^2 - k and generator g = k. This ensures that the public value that C sends in CKE has to be g^c = k. Hence, M can force C to sign a CKE with an arbitrary value k for which it knows the discrete log for the real group chosen by S. Hence, M can finish the handshake with S by computing the master secret and connection keys and then exchange data with S pretending to be C.

Online vs. Offline collisions.

As a proof-of-concept, we computed a collision for two RSA-MD5 client signatures. On a standard desktop, the computation took 24 hours, but the problem is highly parallelizable. With more advanced hardware such collisions can be computed in far less time; the best published result in 2009 took 7 hours for chosen-prefix collisions on academic-scale hardware. Still, a man-in-the-middle attack as described above is not entirely practical (we can mount it on long-lived connections used by Curl and Git, and background connections in Firefox, but not on standard website interactions.)

We note that the collision here depends upon P1 and P2, where the only unknown value is the client random cr. If this value were to be predictable or if it were to be repeated with high frequency, the collision can be computed offline making the impersonation attack practical. The client random can become predictable under some implementation bugs (e.g. see CVE-2015-0285 in OpenSSL); moreover, we note that the TLS standard only requires this value to be unique, not unpredictable. However, our collision attack shows that the client random in TLS needs to be unpredictable to mitigate against signature collisions.

3. COLLIDING TLS 1.3 SERVER AUTHENTICATION

In TLS 1.3, the server hashes and signs the full message transcript in a new ServerCertificateVerify message that occurs immediately after the following sequence:

```
CH CKS SH SKS SC SCR
```

This sequence includes two new kinds of messages:

- ClientKeyShare(CKS): has several client key shares for different (EC)DH groups; we denote the last key share cks.
- ServerKeyShare(SKS): has the server's key share sks for the chosen group.

Suppose C sends CH(cr) CKS(cks) to S. S responds with SH(sr) SKS(sks) SC SCR(auth) to C. The man-in-the-middle M allows the exchange to proceed unmodifies except that it changes SKS(sks) to SKS'(k'), where k' is a chosen public value for which M knows the secret, and it changes SCR(auth) to SCR'(auth'), where auth' is computed as shown below.

M computes a C-byte suffix that causes a hash collision after the following two chosen prefixes:

```
P1 = CH CKS'(cks0)
P2 = CH CKS SH SKS' SC SCR'(auth0)
```

Here, the client key share cks0 is empty but its length is set to C, and the client authority field auth0 is empty but its length is set to C + L where L is the length of the message sequence SH SKS SC SCR(auth) usually sent by S.

Suppose B1 and B2 are the colliding suffixes,
that is Hash(P1 B1) = Hash(P2 B2). Then the hashes are the same for the full transcripts:

```
T1 = P1 B1 SH SKS SC SCR(auth)
T2 = P2 B2 SH SKS SC SCR(auth)
```

where in T2, the sequence SH SKS SC SCR(auth) fits into the auth' field of the original SCR' in P2.

Consequently, an M that gets S to sign T1 can impersonate S at C. As before the attack can be performed offline if the client random were to be predictable or reused.

4. COLLIDING TLS 1.2 SERVER AUTHENTICATION

In the DHE handshake in TLS 1.2, the server's signature is over a message of the form:

```
SKE = cr sr p g gy
```

Here, cr is chosen by the client, and usually sr is chosen by the server. Notably, the size of cr sr is one block of MD5 (64 bytes). What if sr were also chosen by the client? Alternatively, what if cr were large enough to be one MD5 block? Then, a malicious client M can get the server to sign one SKE but send a different one to an honest client C.

Suppose C sends ClientHello(cr) to S. S responds with ServerHello(sr) and the man-in-the-middle M allows the exchange to proceed unmodified until the ServerKeyExchange. M then computes a single-block collision with the following two chosen prefixes:

```
P1 = cr  sr p' g' gy'
P2 = <empty>
```

where `gy'` is an empty bytearray with length set to 64 + L.

Suppose the two MD5 blocks B1 and B2 collide the above prefixes, that is `MD5(P1 B1) = MD5(P2 B2)`. Then the two messages below will have the same hashes:

```
SKE1 = P1 B1 p g gy
SKE2 =    B2 p g gy
```

Note that in `SKE1`, the sequence `B1 p g gy` all fits into `gy'`.

Hence, if M can choose both random values `cr sr = B2`, then it can get S to sign `SKE2` and hence obtain S's signature on `SKE1` for an arbitrary prime and generator. This enables M to impersonate S at C

Practicality of the attack.

In normal TLS, this attack is prevented because the size of the client random is less than one MD5 block size and because the client usually cannot control the server random. However, in recent enhancements to TLS, some designs (e.g. Snap Start) have considered allowing clients to choose the server random, which would then lead to this kind of attack.

Computing a single-block collision is even more time consuming than the multi-block collision of the previous section. As before, if the client random sent by C were predictable, the collision can be found offline.

5. OTHER CONSTRUCTIONS

More generally, collisions in TLS signatures can affect not only DHE but also ECDHE and SRP key exchanges. Hash function collisions also affects the security of the finished messages in TLS, but those messages usually use stronger hash functions than MD5.

We have focused on signature collisions in TLS, but similar collisions occur in other signed Diffie-Hellman protocols such as IKEv2. Challenge-response signature protocols (EAP/SASL) over TLS and IKEv2 are also potentially vulnerable to these kinds of attacks.

In conclusion, we believe that collision resistance is an essential property for hash functions used in popular channel establishment protocols. Second preimage-resistance is not enough.

6. REFERENCES

[1] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[2] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2014.

[3] P. Hoffman. Use of hash algorithms in internet key exchange (ike) and ipsec. Technical report, RFC 4894, 2007.

[4] P. Hoffman and B. Schneier. Attacks on cryptographic hashes in internet protocols. Technical report, RFC 4270, November, 2005.