Popa & Weaver
Spring 2019

CS 161 Computer Security

Discussion 08

Question 1 *DNSSEC / TLS* (15 min)

(a) Oski wants to securely communicate with CalBears.com using TLS. Which of the following entities must Oski trust in order to communicate with confidentiality, integrity, and authenticity?

1. The operators of CalBears.com
2. Oski's computer
3. Cryptographic algorithms
4. Computers on Oski's local network
5. The operators of CalBears.com's authoritative DNS servers
6. The entire network between Oski and CalBears.com
7. CalBears.com's CA
8. All of the CAs that come configured into Oski's browser
9. All of the CAs that come configured into CalBears.com's software
10. The operators of `.com`'s Authoritative DNS servers
11. The operators of the Authoritative DNS root servers

Solution: (1) The operators of CalBears.com, (2) Oski's computer, (3) Cryptographic algorithms, (7) CalBears.com's Certificate Authority, (8) All of the CAs that come configured into Oski's browser. This last would not be the case if Oski's client has pinned the CalBears.com certificate.

(b) Suppose we didn't want to trust any of the existing CAs, but DNSSEC was widely deployed and we were willing to trust DNSSEC and the operators of the root zone and of `.com`. How could TLS be modified, to avoid the need to trust any of the existing CAs, under these conditions?

Solution: The basic idea would be for a TLS client to retrieve a site's public keys via DNSSEC records from the site's domain, rather than via a certificate sent by the server and signed by a CA. Such an approach could also instead return signatures of public keys that the server would then still send to the TLS client; the client would now validate the public key based on the signature received via DNSSEC rather than some CA. The inspiration for this question came from DNS-based Authentication of Named Entities (DANE). DANE is a standard currently under development that, among other things, allows certificates to be bound to DNSSEC records.

(c) Assume end-to-end DNSSEC deployment as well as full deployment of your change. Oski wants to securely communicate with CalBears.com using TLS. What changes are there to the list in part A (i.e., what must Oski trust in order to communicate with confidentiality, integrity, and authenticity)?

Solution: No longer need to trust: (8) All of the CAs that are configured in Oski's browser, (7) CalBears.com Certificate Authority.
Also need to trust: (5) The operators of CalBears.com's authoritative DNS servers, (10) The operators of .com's authoritative DNS servers, (11) The operators of the authoritative DNS root servers.

(d) Is this change good or bad? List at least one positive and one negative effect that would result from this change.

Solution: Many answers are possible here. One could say that it's a good change because there are now fewer parties to trust. Another answer is that it's a good change because it associates trust directly with parties associated with a domain, rather than with all CAs. But one could also argue that now the operators of the root name servers gain a great deal of power.

Question 2 ***NSEC*** **(20 min)**

In class, you learned about DNSSEC, which uses certificate-style authentication for DNS results.

(a) In the case of a negative result (the name requested doesn't exist), what is the result returned by the nameserver to avoid dynamically signing a statement such as "`aaa.google.com` does not exist"? (This should be a review from lecture.)

> **Solution:** The nameserver uses a canonical alphabetical ordering of all record names in its zone. It creates (off-line) signed statements for each pair of adjacent names in the ordering. When a request comes in for which there is no name, the nameserver replies with the record that lists the two existing names just before and just after where the requested name would be in the ordering. This proves the non-existence of the requested name. The reply is called an **NSEC** resource record.
>
> For example, suppose the following names exist in `google.com` when it's viewed in alphabetical order:
>
> ```
> ...
> a-one-and-a-two-and-a-three-and-a-four.google.com
> a1sauce.google.com
> aardvark.google.com
> ...
> ```
>
> In this ordering, `aaa.google.com` would fall between `a1sauce.google.com` and `aardvark.google.com`. So in response to a DNSSEC query for `aaa.google.com`, the name server would return an NSEC RR that in informal terms states "the name that in alphabetical order comes after `a1sauce.google.com` is `aardvark.google.com`", along with a signature of that NSEC RR made using `google.com`'s key.
>
> The signature allows the recipient to verify the validity of the statement, and by checking that `aaa.google.com` would have fallen between those two names, the recipient has confidence that the name indeed does not exist.

(b) One drawback with this approach is that an attacker can now enumerate all the record names in a zone. Why might this be a security concern?

> **Solution:** Revealing this information could aid in other attacks. For example, the names in a zone could be used as targets when probing for vulnerable servers.

(c) Louis proposes to modify NSEC as follows. First, the site operator will take a hash of each domain that does exist. Then, the site operator proceeds as in NSEC: they sort the hashes and sign each adjacent pair. How can this be used to provide authenticated denial? How does this help mitigate enumeration attacks?

Solution: Instead of sorting on the domains, the sorting is done on hashes of the names. For example, suppose the procedure is to use SHA1 and then sort the output treated as hexadecimal digits. If the original zone contained:

```
  barkflea.foo.com
   boredom.foo.com
    bug-me.foo.com
   galumph.foo.com
   help-me.foo.com
perplexity.foo.com
     primo.foo.com
```

then the corresponding SHA1 values would be:

```
  barkflea.foo.com = e24f2a7b9fa26e2a0c201a7196325889abf7c45b
   boredom.foo.com = 6d0edfd3efa5bf11b094cb26a7c95a3bd5e85a84
    bug-me.foo.com = 649bb99765bb29c379d935a68db2eebc95ad6a29
   galumph.foo.com = 71d0549ab66459447a62b639849145dace1fa68e
   help-me.foo.com = 1ed14d3733f88e5794cd30cbbef8cc32fa47db2a
perplexity.foo.com = 446ac4777f8d3883da81631902fafd0eba3064ec
     primo.foo.com = 8a1011003ade80461322828f3b55b46c44814d6b
```

Sorting these on the hex for the hashes:

```
   help-me.foo.com = 1ed14d3733f88e5794cd30cbbef8cc32fa47db2a
perplexity.foo.com = 446ac4777f8d3883da81631902fafd0eba3064ec
    bug-me.foo.com = 649bb99765bb29c379d935a68db2eebc95ad6a29
   boredom.foo.com = 6d0edfd3efa5bf11b094cb26a7c95a3bd5e85a84
   galumph.foo.com = 71d0549ab66459447a62b639849145dace1fa68e
     primo.foo.com = 8a1011003ade80461322828f3b55b46c44814d6b
  barkflea.foo.com = e24f2a7b9fa26e2a0c201a7196325889abf7c45b
```

Now if a client requests a lookup of `snup.foo.com`, which doesn't exist, the name server will return a record that in informal terms states "the hash that in alphabetical order comes after `71d0549ab66459447a62b639849145dace1fa68e` is `8a1011003ade80461322828f3b55b46c44814d6b`" (again along with a signature made using `foo.com`'s key). This type of Resource Record is called **NSEC3**.

The client would compute the SHA1 hash of `snup.foo.com`:

```
      snup.foo.com = 81a8eb88bf3dd1f80c6d21320b3bc989801caae9
```

and verify that in alphabetical order it indeed falls between those two returned values (standard ASCII sorting collates digits as coming before letters). That confirms the non-existence of `snup.foo.com` but without indicating what names do exist, just what hashes exist.

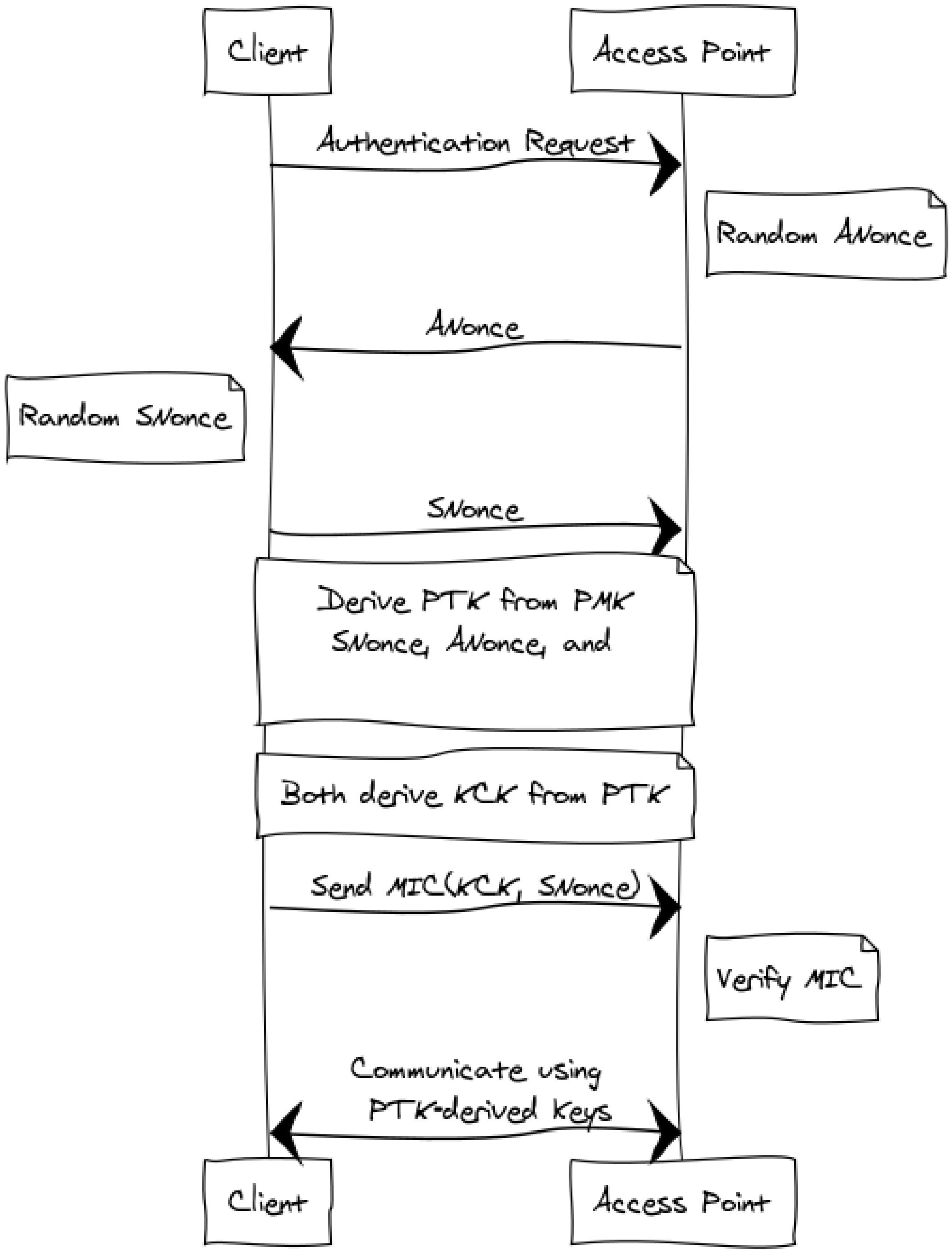
By using a cryptographically strong hash function ~~like SHA1~~[1], it's believed infeasible to reverse the hash function to find out what name(s) appear in the zone (there's more than one potential name because hash functions are many-to-one). Note though that an attacker can still conduct a dictionary attack, either directly trying names to see whether they exist, or inspecting the hash values returned by NSEC3 RRs to determine whether names in a dictionary (for which the attacker computes hash values offline) indeed appear in the domain.

[1]As we know, SHA1 is no longer considered secure for many use cases. Using stronger hash functions for DNSSEC is therefore recommended. That said, the property we need from the hash function is one-way-ness, which to date is not an identified weakness of SHA1 (nor of MD5, in fact).

Question 3 *WPA2* (15 min)

Let's review WPA2. You might find some of the definitions below helpful.

- PMK is the *premaster key*, also known as "the WiFi password".
- PTK is the *pairwise transient key*, which is used to derive symmetric keys.
- KCK is the *key confirmation key*, which helps the client and the access point confirm they've agreed on the same keys.



(a) Louis Reasoner proposes that we don't generate ANonce or SNonce, and instead derive the PTK directly from the SSID and PMK. What sort of attack does this fail to prevent?

> **Solution:**
>
> Replay attacks! Nonces always stop replay attacks.

(b) WPA2 has an interesting pattern which is common in cryptographic protocols. Both parties agree on a shared secret, which they use to derive keys. Which other protocol have we seen which follows this motif?

> **Solution:**
>
> TLS–both parties agree on a premastered secret.

(c) Alyssa P. Hacker wants to compromise a WPA2 WiFi network. In order to do so, she performs the handshake many times. She bruteforces possible PMK against the Access Point many times, until the access point eventually accepts it. If the password has 28 bits of entropy[2] and the attacker can make 10 guesses a second, how long will it take to bruteforce the password?

> **Solution:**
>
> $2^{28}/10 \approx 310$ days.

(d) Ben Bitdiddle has an alternate idea. Ben waits until Louis attempts to connect to the network. While this happens, he records all of the messages that Louis sends over the network. How can Ben use this to bruteforce possible PMKs? Why do we expect this to be faster than Alyssa's method?

> **Solution:**
>
> Ben can attempt to bruteforce PMK and derive keys the same way the access point and Louis would. Once he gets the same key confirmation key (which he can check by looking at a MIC computed with the key-confirmation key), then he knows that he's probably generated the same keys and hence has the right PMK.
>
> This is significantly faster than Alyssa's method because it can be computed offline.

[2] As per this XKCD comic, a password which looks like `Tr0ub4dor&3` has roughly 28 bits of entropy.