Paxson
Spring 2017

CS 161
Computer Security

Discussion 11

Week of April 17, 2017

Question 1 ***DNSSEC*** **(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 is this 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) How could you change the response sent by the nameserver to avoid this issue?

HINT: One of the crypto primitives you learned about will be helpful.

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.

Question 2 *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

[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).

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 3 *TLS downgrade attacks* (15 min)

(a) Rather than prescribing specific cryptographic functions, the TLS protocol allows the browser and server to agree on a cipher suite. What are the different components of a cipher suite that the parties need to negotiate?

Solution: The client and server need to negotiate the functions for each of the cryptographic operations performed by TLS. In particular, they need to agree about:

- The **key exchange method**, such as RSA or Diffie-Hellman. In the latter case, an **authentication method** is also needed to specify how the server will use the key in its certificate to authenticate the Diffie-Hellman parameter (i.e., which signature algorithm it will use).
- The **encryption algorithm** used on the data sent over the secure channel. Specifying this typically requires including the **key size** and **cipher mode** to use with the specific encryption function.
- The **MAC algorithm** used to authenticate the data.
- The **pseudorandom function** used to derive the master secret (essentially, a PRNG).

A specific combination of these choices is called a *cipher suite*, and there are hundreds to choose from. Each is referred to by a name like `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`, which represents some of the chosen crypto functions (though to get the full details, you have to look it up in the appropriate RFC).

(b) How do the parties find out which cipher suites the other supports? Who ultimately gets to choose which cipher suite is chosen? How do they choose?

Solution: The client includes a list of the cipher suites it supports in the `ClientHello` message.

The server will determine which of the client's cipher suites it supports, decide (assuming there is overlap) which one it wants to use, and notify the client of its choice as part of the `ServerHello` message.

A server will typically want to use the strongest cipher suite and parameters available, however this is determined entirely by the server's configurations. It is therefore very important to configure a server's TLS carefully.

(c) Suppose Mallory knows how to break certain cryptographic primitives. Alice and Bob.com, communicating over TLS, both support the cipher suite with the vulnerable crypto, as well as others that are not broken. How can Mallory carry out a man-in-the-middle attack? Which of the crypto primitives must she be able to break for the attack to succeed?

Solution: When Alice sends her `ClientHello` to Bob.com, she will include the weak cipher suites as well as the ones that aren't vulnerable. By default, Bob.com is likely to choose one of the stronger suites, but Mallory can force his hand.

Since the `ClientHello` isn't encrypted, Mallory can change it to exclude the cipher suites she doesn't want Bob.com to choose. For example, if she knows how to break 3DES encryption, she can remove all cipher suites that use other encryption algorithms. This is called a *downgrade attack*.

However, you'll recall that TLS includes a step to verify the integrity of the handshake: as part of the `Finished` message, the client and server exchange MACs of their prior dialog. Since this will include the `ClientHello` message and its contents, Mallory needs to be able to falsify this message. Therefore, if she wants to carry out a successful man-in-the-middle attack, Mallory must be able to break the MAC function, in addition to whatever else she was able to break.

Bonus fun fact Interestingly, two known downgrade attacks on TLS—FREAK and Logjam—were able to circumvent this requirement. Both targeted the key exchange algorithm and relied on downgrading connections from secure versions of RSA and Diffie Hellman to "export" versions that used smaller key sizes and were consequently broken.

In these attacks, a man-in-the-middle could modify the client's `Hello` to ask for weaker parameters. When the server responded, the client couldn't tell that it had received weaker parameters due to bugs in client implementations or TLS itself[2](in FREAK and Logjam, respectively).

While the later MACs over the handshake would normally catch the discrepancy, by that point, it's too late: with the weakened key-exchange parameters, the attacker can compromise the encrypted premaster secret to derive the master secret. Knowing the master secret, the attacker can generate their own fake MACs (as well as decrypt any future communication).

(d) Is there anything Alice and Bob.com can do to prevent attacks like this?

Solution: Fundamentally, no: as long as both parties are willing to use weak cryptographic primitives, and these are broken (including, specifically, the ones that guarantee integrity, as described above), then a downgrade attack is theoretically possible.

However, as the real-world examples show, many attacks can be prevented in

[2] The attack "relies on a flaw in the way TLS composes DHE and DHE_EXPORT. When a server selects DHE_EXPORT for a handshake, it proceeds by issuing a signed ServerKeyExchange message containing a 512-bit p_{512}, but the structure of this message is identical to the message sent during standard DHE ciphersuites. Critically, the signed portion of the server's message fails to include any indication of the specific ciphersuite that the server has chosen. [...] The client will interpret the export-grade tuple (p_{512}, g, g^b) as valid DHE parameters chosen by the server and proceed with the handshake."

practice by being explicit about one's choices and including as much information as possible when checking for integrity.