# Chapter 4

## Authentication Protocols and Key Establishment



The official version of this book is available at
https://www.springer.com/gp/book/9783030336486

ISBN: 978-3-030-33648-6 (hardcopy), 978-3-030-33649-3 (eBook)



version: 25 Sept 2019

# Chapter 4

# Authentication Protocols and Key Establishment

This chapter discusses authentication protocols involving cryptographic algorithms. The main focus is *authenticated key establishment* protocols seeking to establish a cryptographic key (secret) for subsequent secure communications, with assurance of the identity of the far-end party sharing the key. Several mainstream key establishment protocols are discussed, as well as examples of what can go wrong. We also discuss *password-authenticated key exchange*, designed to resist offline attacks even if users choose predictable passwords, as well as *single sign-on* (SSO) systems and related *federated identity systems*. A main objective is to highlight that even experts find it hard to avoid subtle errors in the design of authentication protocols; software designers should use standardized protocols and carefully scrutinized software libraries, and fully expect that any protocols they design themselves will almost surely contain hidden flaws.

As context, recall that Chapter 3 discussed *dictionary attacks* that used password hashes to test guesses offline, in order to recover user-chosen passwords. Whereas there the hashes were from stolen password files, here we discuss related attacks that instead recover *weak secrets* (keys derived from user-chosen passwords) by offline testing using data recorded from messages exchanged in vulnerable key establishment protocols.

## 4.1 Entity authentication and key establishment (context)

We begin with some context and definitions. A *protocol* is an exchange of messages between parties (devices). *Entity authentication* is a process to verify the identity of a communicating party. A *cryptographic protocol* is a protocol that involves cryptographic techniques (e.g., beyond sending a password itself). An *authentication protocol* is a cryptographic protocol that provides entity authentication, authenticated key establishment (below), or both. Figure 4.1 first explains basic claimant-verifier authentication.

**Example** *(Browser-server authentication).* Typical web browser-server authentication involves the server (as *claimant* or *prover*) providing evidence to convince the browser

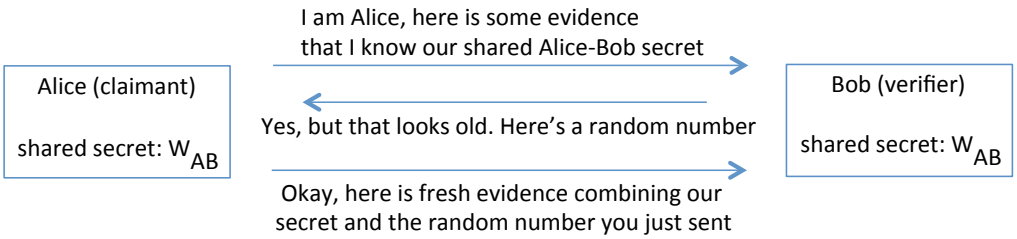



Figure 4.1: Basic unilateral authentication. The *claimant* is the party (entity or device) being authenticated. The party given assurances is the *verifier*. We may use $W$ to denote a weak (password-based) secret, and $S$ to denote a crypto-strength random key.

of the server's legitimacy. This is called *unilateral authentication*, with one party authenticating itself to another. In *mutual authentication*, each party proves its identity to the other; this is largely unused in the standard web protocol (TLS), despite being supported. If authentication of the browser (user) to the server is desired, this is commonly done by password-based authentication using an encrypted channel set up in conjunction with the unilateral authentication. Aside: when a credit card is used for a web purchase, the server typically does not carry out authentication of the user *per se*, but rather seeks a valid credit card number and expiry date (plus any other data mandated for credit approval).

**SESSION KEYS.** *Key establishment* is some means by which two end-parties arrange a shared secret—typically a symmetric key, i.e., a large random number or bitstring—for use in securing subsequent communications such as client-server data transfer, or voice/video communication between peers. Such keys used for short-term purposes, e.g., a communications session, are called *session keys*, or *data keys* (used for encrypting data, rather than for managing other keys). Key establishment has two subcases, discussed next.

**KEY TRANSPORT VS. KEY AGREEMENT.** In *key transport*, one party unilaterally chooses the symmetric key and transfers it to another. In *key agreement*, the shared key is a function of values contributed by both parties. Both involve leveraging long-term keying material (shared secrets, or trusted public keys) to establish, ideally, new *ephemeral keys* (secrets that are unrecoverably destroyed when a session ends). Key agreement commonly uses variations of Diffie-Hellman (Section 4.3) authenticated by long-term keys. If session keys are instead derived deterministically from long-term keys, or (e.g., RSA) key transport is used under a fixed long-term key, then compromise of long-term keys puts at risk all session keys (see *forward secrecy*, Section 4.4). Figure 4.2 relates types of authentication and key establishment algorithms, and cryptographic technologies.

**AUTHENTICATION-ONLY, UNAUTHENTICATED KEY ESTABLISHMENT.** Some protocols provide assurances of the identity of a far-end party, without establishing a session key. Such *authentication-only protocols*—named to avoid confusion with authenticated key establishment protocols—may be useful in restricted contexts. An example is local authentication between your banking chip-card and the automated banking machine you are standing in front of and just inserted that card into. But if authentication-only occurs across a network at the beginning of a communications session, a risk is that the

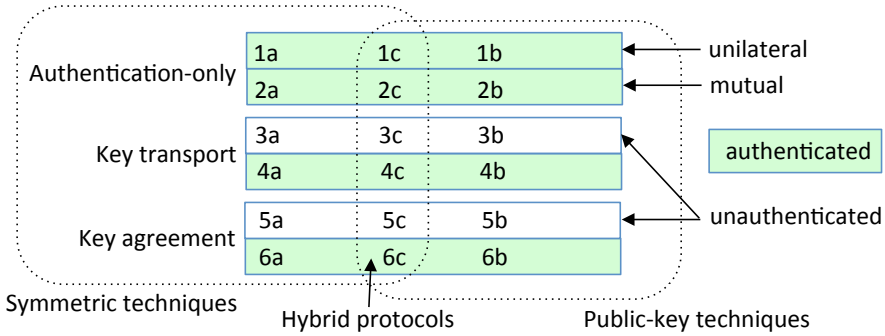



Figure 4.2: Authentication and key establishment protocol taxonomy. Hybrid protocols combine symmetric-key and public-key techniques. Example protocols and categories: basic Diffie-Hellman (5b), STS (6c), DH-EKE (6c), SPEKE (6b), and Kerberos (4a).

session is "hijacked" after start-of-session authentication, with subsequent data transfer directed to a different party.[1] Whether this is possible depends on the communication channels and networking protocols used, but is an issue to consider in deciding whether an authentication-only protocol suits a given application.

Other protocols establish a shared session key with a remote second party, with no assurances or guarantees as to the identity of that second party. An example of such *unauthenticated key establishment* is basic Diffie-Hellman (Section 4.3). This may be fine against passive attackers (eavesdroppers), but as we will see, problems arise in the case of active attackers. While it may seem obvious, we state explicitly: if you establish a shared key with a second party, but don't have explicit assurances as to the identity of that party, you may be communicating with (or through) a different party than you believe.

**INTEGRATING AUTHENTICATION WITH SESSION KEY ESTABLISHMENT.** If key establishment is to provide some assurance of whom an established key is shared with, it must involve some means of authentication. Experience has shown that using separate key establishment and entity authentication protocols, and then trying to glue them together, tends to end badly; pursuing both functions within one integrated protocol appears necessary to ensure that the party authenticated is the same party that the key is shared with. Such a combined process is called *authenticated key establishment*. Using the established session key for ongoing integrity can then be thought of as "keeping authentication alive".

**KEY MANAGEMENT.** Authentication involving cryptographic protocols naturally relies on cryptographic keys. In practice, the most challenging problem is *key management*: establishing shared keys, securing them in transit and in storage, and for public keys, establishing trust in them and maintaining their integrity and authenticity. Keys used for securing data at rest (storage) need not be shared with a second party, but it is important to have some means to recover if such keys are lost, e.g., due to hardware failure (without a key backup, expect to lose access to all encrypted data, permanently). For keys to be used in communications, things are better and worse: consequences are less severe if a session

[1] This may be done, e.g., by exploiting authentication weaknesses in TCP, as explained in Chapter 11.

key is lost (just establish a new key and retransmit), but shared keys must be arranged between sender and recipient, e.g., to transmit encrypted data.

**RE-USING SESSION OR DATA KEYS.** For various reasons it is poor cryptographic hygiene to use permanent (static) session or data keys; to re-use the same such keys with different parties; and to re-use session or data keys across different devices. Every place a secret is used adds a possible exposure point. The greater the number of sessions or devices that use a key, the more attractive a target it becomes. History also shows that protocols invulnerable to attacks on single instances of key usage may become vulnerable when keys are re-used. Secrets have a tendency to "leak", i.e., be stolen or become publicly known. Secrets in volatile memory may dump to disk during a system crash, and a system backup may then store the keys to a cloud server that is supposed to be isolated but somehow isn't. (Oops.) Implementation errors result in keys leaking from time to time. (Oops.) For these and other reasons, it is important to have means to generate and distribute new keys regularly, efficiently and conveniently.

**INITIAL KEYING MATERIAL.** To enable authenticated key establishment, a *registration phase* is needed to associate or distribute initial keying material (public or secret) with identified parties. This usually involves *out-of-band* means.[2] This is usually code for: find some way to establish shared secrets, or transfer data with guaranteed authenticity (integrity), either without using cryptography, or using independent cryptographic mechanisms—and often by non-automated processes or manual means. For example, we choose a password and "securely" share the password (or PIN) with our bank in person, or the bank sends us a PIN by postal mail, or some existing non-public information is confirmed by phone. User-registered passwords for web sites are sent over TLS encryption channels secured by the out-of-band means of the browser vendor having embedded CA public keys into the browser software.

**CRYPTO-STRENGTH KEYS, WEAK SECRETS.** Ideally, symmetric keys for use in crypto algorithms are generated by properly seeded cryptographic random number generators such that the keys are "totally random"—every possible secret key (bitstring) is equally likely. Then, if the game is to search for a correct key, no strategy is better than an exhaustive enumeration of the key space, i.e., of all possible values. For keys of $t$ bits, an attacker with a single guess has a 1 in $2^t$ chance of success, and no strategy better than a random guess. An attacker enumerating the full key space can on average expect a correct guess after searching half the space, e.g., $2^{127}$ keys for $t = 128$. Choosing $t$ large enough makes such attacks infeasible. We call secrets chosen at random, and from a sufficiently large space, *crypto-strength keys* or *strong secrets*. In contrast a key generated deterministically by hashing a user-chosen password is a *weak secret*. Sections 4.5 and 4.6 explore weak secrets and how protocols can fail when they are used in place of strong secrets.

**Exercise** (Protecting long-term keys). How do we protect long-term secrets stored in software? Discuss. Likewise consider short-term secrets (passwords and keys).

‡**POINT-TO-POINT MODEL WITH $n^2$ KEY PAIRS.** Each pair of parties should use a unique symmetric key to secure communications. Given $n$ communicating parties, this

[2] *Out-of-band* means are also discussed in Chapter 8.

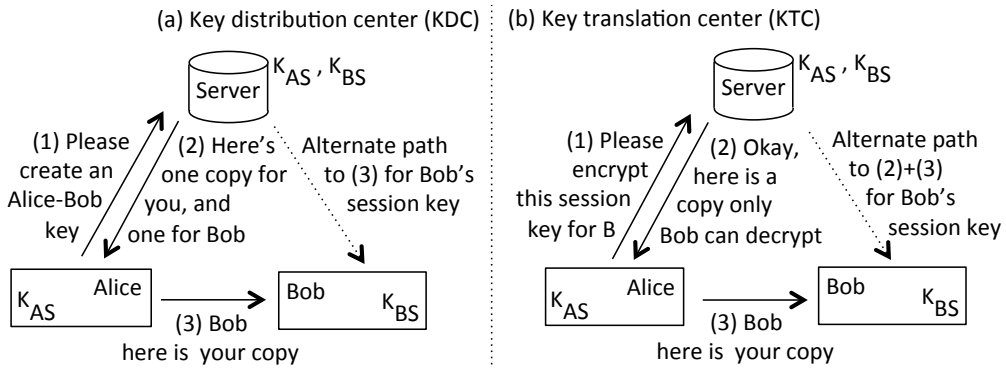



Figure 4.3: Key transport models for centralized symmetric-key distribution. (a) KDC: the trusted server generates the session key $K_{AB}$. (b) KTC: end-party $A$ generates the session key, but needs the server to encrypt it for $B$. The server distributes session keys (as the hub in a hub-and-spoke architecture) because end-parties don't share long-term pairwise keys with each other; each shares a long-term key only with the server. $K_{AS}$ is a long-term secret (key) shared by $A$ (Alice) and $S$; $K_{BS}$ is likewise for $B$ (Bob) and $S$.

means ($n$ choose 2) $= n(n-1)/2 \approx n^2$ pairs of parties, or $n^2$ overall keys in such a *point-to-point* network model. Each party would have $n-1$ keys, one for each potential communicating partner. An organization may need to back up $n^2$ keys, and/or update them regularly, and securely distribute $n^2$ pairs of keys, e.g., by manual means. The cost of this grows rapidly with $n$. This motivates use of a centralized hub-and-spoke model with a server at the center; this is done (albeit in different ways) both in systems using symmetric-key techniques (here), and public-key techniques (Chapter 8).

‡CENTRALIZED SYMMETRIC-KEY SERVERS: KDC, KTC. In symmetric-key technology systems, a *centralized* architecture reduces the $n^2$ complexity of key distribution by using a *trusted server* $S$ as follows. Each party $A$ has one unique, long-term symmetric key that it shares with $S$, but no long-term keys shared with other parties; as notation, let $K_{AS}$ denote the $A$-$S$ such key. The basic idea is as follows, with two options (Fig. 4.3). One is a *key distribution center* (KDC).[3] Party $A$ makes a request asking $S$ to set up a pairwise $A$-$B$ session key. $S$ generates such a key $K_{AB}$, and sends it to $A$ and $B$ separately, respectively symmetrically encrypted under $K_{AS}$ and $K_{BS}$, as $\{K_{AB}\}_{K_{AS}}$ and $\{K_{AB}\}_{K_{BS}}$; alternately $S$ could send $B$'s copy to $A$ directly, for $A$ to forward on to $B$.

A second option is a *key translation center* (KTC). Here session key $K_{AB}$ is generated by end-party $A$, which sends it to $S$ encrypted as $\{K_{AB}\}_{K_{AS}}$. $S$ decrypts the session key, re-encrypts it for $B$ under long-term key $K_{BS}$ as $\{K_{AB}\}_{K_{BS}}$, and sends it to $B$ (or alternately, sends it back to $A$ to forward on to $B$). In summary, both options reduce the $n^2$ key distribution complexity using symmetric technology and a trusted online centralized server. A KDC chooses all session keys; a KTC has end-parties choose session keys. The long-term keys (initial keying material) each party shares with $S$ are set up by *out-of-band*

[3] A specific example is the Kerberos system in Section 4.7, which includes important details omitted here.

techniques, e.g., in-person exchanges, or use of couriers trusted to protect the confidentiality of the keying material.

**CHOICE OF SYMMETRIC-KEY OR PUBLIC-KEY METHODS.** Either symmetric-key or public-key approaches (Chapter 8), or their combination, can be used for entity authentication and authenticated key establishment. As discussed next, in both cases, designing security protocols resistant to creative attacks has proven to be quite challenging, and provides an example of principle P9 (TIME-TESTED-TOOLS).

## 4.2 Authentication protocols: concepts and mistakes

Here we consider basic concepts about authentication protocols, along with illustrative protocol fragments. Discussion of symmetric-key and public-key protocols is combined, to address issues common to both. Detailed discussion of Diffie-Hellman and related key establishment protocols is in later sections; the Kerberos protocol, which provides authenticated key establishment using symmetric keys, is discussed in Section 4.7.

**DEMONSTRATING KNOWLEDGE OF SECRET AS PROXY FOR IDENTITY.** A basic idea used to authenticate a remote party $B$ is to (a) associate a secret with $B$; and then (b) carry out a communication believed to be with $B$, accepting a demonstration of knowledge of that secret (key) as evidence that $B$ is the party involved in the communication. As discussed (page 94), it is desirable to combine this entity authentication with establishing a session key for that communication session, distinct from authentication keys. There are many ways this can go wrong, which makes design of authentication protocols more complicated than it seems. This is so even without violation of the base assumption that the authentication secret has not been shared with other parties (or stolen).

If the demonstration of knowledge involves revealing the full secret itself (such as a password to a server), then it should be over a channel guaranteeing confidentiality, and such a channel must be first set up to rely on. Rather than send the secret itself, it is preferred to send (convincing) evidence of knowledge of the secret, informally called a *proof of knowledge*. Let's try the first thing that comes to mind, and see what goes wrong. While our main interest is authenticated key establishment, our flawed-protocol examples here will use authentication-only protocols, since they are simpler and allow the lessons to be learned more easily—and the same conceptual flaws often apply to both.

**SIMPLE REPLAY ATTACK.** Suppose the secret shared by $A$ and $B$ is $S$. $A$ wants to prove knowledge of $S$. To avoid revealing $S$ directly, $A$ sends $H(S)$, a one-way hash of $S$, thinking: this doesn't reveal $S$—and B (Bob) can take his own copy of $S$, the same known hash function, compute $H(S)$, and compare that to what is received. However, the attacker simply needs to capture and replay $H(S)$, without ever knowing $S$ itself. This is a simple *replay attack*. The flaw is that replays of parts of old protocol runs defeat the protocol.

**DICTIONARY ATTACK ON WEAK SECRET.** The above attack suggests the following simple *challenge-response* protocol. We'll again give an example of what can go wrong. Hash function $H$ is as above, but now let's assume a weak shared secret $W$. In this case $B$ will be the claimant. $A$ sends $B$ a fresh, long, random number $r_A$. She expects back the

hashed concatenation of $r_A$ and $W$, which $A$ will compare to a similarly computed value using her saved copy of $r_A$, and $W$:

(1) $A \rightarrow B:\ A, r_A$ ... Alice here, prove to me you are Bob using our shared $W$

(2) $A \leftarrow B:\ H(r_A, W)$ ... this hash of $W$ with fresh $r_A$ should convince you I'm Bob

What can go wrong? If $W$ is a weak secret (e.g., password), then this provides an attacker *verifiable text* against which an *offline guessing attack* can be mounted to recover $W$. It's like a password hash-file dictionary attack, without having to steal the file of hashed passwords. A stronger secret $W$ is apparently needed, but next we see that is not enough.

**REFLECTION ATTACK.** The following *reflection attack* succeeds even if the shared secret above is a crypto-strength key. $A$ and $B$ now share $W = S = S_{AB}$. Alice sends (1) above to Bob; Bob is away on vacation but attacker $C$ impersonates $B$ to $A$ as follows. $C$ starts a new protocol run in parallel, sending $A$ a new message (1)* substituting in identifier $B$ (claiming to be Bob) but re-using the same $r_A$: $C \rightarrow A : B, r_A$. Since $W = S_{AB}$ is the secret $A$ and $B$ use to authenticate each other, $A$'s response $H(r_A, W)$ is exactly the response that $C$ can play back immediately on $A$ to answer $A$'s (1) in the first protocol run.

‡**Exercise** (Mitchell's reflection attack). Describe a reflection attack similar to the above, on a symmetric-key mutual authentication protocol (hint: see Attack 2 [33, p.530]).

**RELAY ATTACK.** An automobile executive notices the above challenge-response protocol, and decides this would make an excellent basis for conveniently unlocking car-door locks—the door will unlock when in sufficient proximity to a low-power wireless (RF/radio frequency) challenge-response between door and owner-carried keyfob. But a *relay attack* defeats the simple challenge-response protocol even if $W$ is replaced by a strong key. The trick is to capture the signal at one place, and relay it in real time to another. So using, e.g., a directional antenna, signal booster and relay, the RF signal between the keyfob and car door can be manipulated so that they appear to be co-located. Voilà.

**IFF RELAY.** Challenge-response authentication dates to 1950-era military *identify-friend-or-foe* (IFF) systems. When sent a random challenge, an aircraft keyed with a (friendly) key could send a correct response by encrypting the challenge with that key. A potential relay attack is as follows: a hostile aircraft receiving a challenge sends it in turn to a friendly aircraft, and uses that response. A critical issue is whether the relay round trip is fast enough to meet the delay allowed by the challenging party. Relay attacks exploit the transferability of responses. *Distance-bounding protocols* may be custom-designed with tight time-delay tolerances (taking into account expected attacker technologies) aiming to preclude relay attacks beyond prescribed distances. The grandmaster chess problem (below) is an instance of a relay attack benefiting from lack of a time bound.

‡**Exercise** (Interleaving attack). An *interleaving attack* on an authentication protocol is an attempt to impersonate, extract keying material, or otherwise defeat the protocol goals by using messages or parts from one or more previous protocol runs, or currently ongoing protocol runs (in parallel sessions), possibly including attacker-originated protocol runs. Explain the technical details of such an attack on the authentication-only protocol discussed in "Attack 3" [33, p.531] (see further discussion in Diffie [14]).

**ATTACKER GOALS.** Table 4.1 summarizes common attacks on authentication protocols. The table and caption suggest attack approaches and strategies. Common attacker

| Attack | Short description |
|---|---|
| replay | re-using a previously captured message in a later protocol run |
| reflection | replaying a captured message to the originating party |
| relay | forwarding a message in real time from a distinct protocol run |
| interleaving | weaving together messages from distinct concurrent protocols |
| middle-person | exploiting use of a proxy between two end-parties |
| dictionary | using a heuristically prioritized list in a guessing attack |
| forward search | feeding guesses into a one-way function, seeking output matches |
| pre-capture | extracting client OTPs by social engineering, for later use |

Table 4.1: Some common attacks on authentication protocols. Attackers follow no rules, and may read, alter, re-use old, and send entirely new messages; and originate new protocol runs, recombining old with current or new. A common *threat model* of prudent protocol designers is that all bitstrings pass through an attacker-controlled point; forwarding a bitstring unchanged does not constitute a successful attack. Defenses include use of *time-variant parameters*, e.g., to cryptographically bind messages within a protocol run.

end-goals (as opposed to approaches) include: to impersonate another party (with or without gaining access to a session key); to discover long-term keys or session keys, either passively or by active protocol manipulation; and to mislead a party as to the identity of the far-end party it is communicating with or sharing a key with.

**CHALLENGE-RESPONSE PROTOCOLS, TIME-VARIANT PARAMETERS.** A few attacks in Table 4.1 rely on re-using messages from previous or ongoing protocol runs. As a defense, *time-variant parameters* (TVPs) provide protocol messages and/or session keys uniqueness or timeliness (freshness) properties, or cryptographically bind messages from a given protocol run to each other, and thereby distinguish protocol runs. Three basic types of TVPs are as follows (each may also be referred to as a *nonce*, or number used only once for a given purpose, with exact properties required depending on the protocol).

1. *random numbers*: In challenge-response protocols, these are used to provide freshness guarantees, to chain protocol messages together, and for conveying evidence that a far-end party has correctly computed a session key (*key-use confirmation*, Section 4.4). They also serve as *confounders* (Section 4.6) to stop certain types of attacks. They are expected to be unpredictable, never intentionally re-used by honest parties, and sufficiently long that the probability of inadvertent re-use is negligible. If a party generates a fresh random number, sends it to a communicating partner, and receives a function of that number in a response, this gives assurance that the response was generated after the current protocol run began (not from old runs).
2. *sequence number*: In some protocols, message uniqueness is the requirement, not unpredictability. A sequence number or monotonic counter may then be used to efficiently rule out message replay. A real-life analogue is a cheque number.
3. *timestamps*: Timestamps can give timeliness guarantees without the challenge part of challenge-response, and help enforce constraints in time-bounded protocols. They require synchronized clocks. An example of use is in Kerberos (Section 4.7).

**RSA ENCRYPTION USED FOR KEY TRANSPORT.** Key agreement using public-key methods is discussed in Section 4.3. Key transport by public-key methods is also common. As an example using RSA encryption, one party may create a random symmetric key $K$, and encrypt it using the intended recipient $B$'s encryption public key: $A \rightarrow B : E_B(K)$. The basic idea is thus simple. Some additional precautions are needed—for example use as just written is vulnerable to a replay attack (to force re-use of an old key), and gives no indication of who sent the key.

**Example** *(RSA decryption used for entity authentication).* Consider:

(1) $A \rightarrow B$ : $H(r_A)$, $A$, $E_B(r_A, A)$ ... $E_B(r_A, A)$ is a public-key encrypted challenge
(2) $A \leftarrow B$ : $r_A$ ... $H(r_A)$ showed knowledge of $r_A$, not $r_A$ itself

Here $r_A$ is a random number created by $A$; $H$ is a one-way hash function. Receiving (1), $B$ decrypts to recover values $r_A^*$, $A^*$; hashes to get $H(r_A^*)$, and cross-checks this equals the first field received in (1); and checks that $A^*$ matches the cleartext identifier received. On receiving (2), $A$ checks that the received value equals that sent in (1). The demonstrated ability to do something requiring $B$'s private key (i.e., decryption) is taken as evidence of communication with $B$. The association of $B$ with the public key used in (1) is by means outside of the protocol. If you find all these checks, their motivations, and implications to be confusing, that is the point: such protocols are confusing and error-prone.

‡**Example** *(HTTP digest authentication).* HTTP *basic access authentication* sends cleartext username-password pairs to a server, and thus requires pairing with encryption, e.g., HTTP with TLS as in HTTPS (Chapter 8). In contrast, HTTP *digest access authentication* uses challenge-response: the client shows knowledge of a password without directly revealing it. A hash function $H$ (e.g., SHA-256) combines the password and other parameters. We outline a simplified version. The client fills a server form with hash value

$$H(h_1, \texttt{Snonce}, \texttt{Cnonce}), \quad \text{where } h_1 = H(\texttt{username}, \texttt{realm}, \texttt{pswd}),$$

along with the client nonce `Cnonce`. The server has sent the nonce `Snonce`, and a string `realm`, describing the host (resource) being accessed. This may help the client determine which credentials to use, and prevents $h_1$ (if stolen from a password hash file) from being directly used on other realms with the same username-password; servers store $h_1$. `Cnonce` prevents an attacker from fully controlling the value over which a client hash is computed, and also stops pre-computed dictionary attacks. This digest authentication is cryptographically weak: it is subject to offline guessing due to verifiable text (Section 4.5; it thus should be used with HTTPS), and uses the deprecated approach of secret data input (here a password) to an unkeyed hash $H$, rather than using a dedicated MAC algorithm.

‡**Exercise** (.htdigest file). To verify HTTP digest authentication, an Apache web server file `.htdigest` store lines "*user*:*realm*:$h_1$" where $h_1 = H(user, realm, pswd)$. A corresponding htdigest shell utility manages this file. Describe its command-line syntax.

## 4.3 Establishing shared keys by public agreement (DH)

We now discuss Diffie-Hellman key agreement, ElGamal encryption, and STS, a protocol that adds mutual authentication to DH. Section 4.8 gives helpful math background.

**DIFFIE-HELLMAN KEY AGREEMENT.** *Diffie-Hellman key agreement* (DH) was invented in 1976. It allows two parties with no prior contact nor any pre-shared keying material, to establish a shared secret by exchanging numbers over a channel readable by everyone else. (Read that again; it doesn't seem possible, but it is.) The system parameters are a suitable large prime $p$ and generator $g$ for the multiplicative group of integers modulo $p$ (Section 4.8); for simplicity, let $g$ and $p$ be fixed and known (published) as a one-time set-up for all users. Modular exponentiation is used.

(1) $A \to B$ : $g^a \pmod p$ ... $B$ selects private $b$, computes $K = (g^a)^b \bmod p$

(2) $A \leftarrow B$ : $g^b \pmod p$ ... $A$ uses its private $a$, computes $K = (g^b)^a \bmod p$

The private keys $a$ and $b$ of $A$, $B$ respectively are chosen as fresh random numbers in the range $[1, p-2]$. An attacker observing the messages $g^a$ and $g^b$ cannot compute $g^{ab}$ the same way $A$ and $B$ do, since the attacker does not know $a$ or $b$. Trying to compute $a$ from $g^a$ and known parameters $g, p$ is called the *discrete logarithm problem*, and turns out to be a difficult computational problem if $p$ is chosen to have suitable properties. While the full list is not our main concern here, $p$ must be huge and $p-1$ must have at least one very large prime factor. The core idea is to use discrete exponentiation as a one-way function, allowing $A$ and $B$ to compute a shared secret $K$ that an eavesdropper cannot.

‡**POSTPROCESSING BY KDF.** Regarding the DH key $K$ here and similarly with other algorithms, for security-related technical reasons, in practice $K$ is used as input to a *key derivation function* (KDF) to create the session key actually used.

**Exercise** (Diffie-Hellman toy example). For artificially small parameters, e.g., $p = 11$ and $g = 2$, hand-compute (yes, with pencil and paper!) an example Diffie-Hellman key agreement following the above protocol description. What is your key $K$ shared by $A$, $B$?

‡**ELGAMAL ENCRYPTION.** A variation of DH, called *ElGamal encryption*, may be used for *key transport*. Assume all parties use known $g$ and $p$ as above. Each potential recipient $A$ selects a private key $a$ as above, computes $g^a \bmod p$, and advertises this (e.g., in a certificate) as its (long-term) public key-agreement key. Any sender $B$ wishing to encrypt for $A$ a message $m$ ($0 \le m \le p-1$, perhaps containing a session key) obtains $g^a$, selects a fresh random $k$ ($1 \le k \le p-2$), and sends:

$B \to A$ : $c = (y, d)$, where $y = g^k \bmod p$, and $d = m \cdot (g^a)^k \bmod p$.

To recover $m$, $A$ computes: $t = y^{p-1-a} \bmod p$ (note this equals $y^{-a} \equiv (g^k)^{-a} \equiv g^{-ak}$). Then $A$ recovers: $m = d \cdot t \bmod p$ (note $d \cdot t \equiv m \cdot g^{ak} \cdot g^{-ak}$). In essence, the DH key $g^{ak}$ is immediately used to transfer a message $m$ by sending the quantity $m \cdot g^{ak} \bmod p$. (Note: each value $k$ encrypts a fixed value $m$ differently; this is an instance of *randomized encryption*. For technical reasons, it is essential that $k$ is random and not re-used.)

**TEXTBOOK DH MEETS SMALL-SUBGROUP ATTACKS.** DH key agreement as outlined above is the "textbook" version. It gives the basic idea. Safe use in practice requires additional checks as now discussed. If an attacker substitutes the value $t = 0$ for exponentials $g^a$ and $g^b$, this forces the resulting key to 0 (confirm this for yourself); not a great secret. Things are similarly catastrophic using $t = 1$. These seem an obvious sort of thing that would be noticed right away, but computers must be instructed to look. We should also rule out $t = p-1 = -1 \bmod p$, since using that as a base for later exponentiation can generate only 1 and $-1$ (we say $t = -1 \bmod p$ generates a *subgroup of order* 2). Perhaps

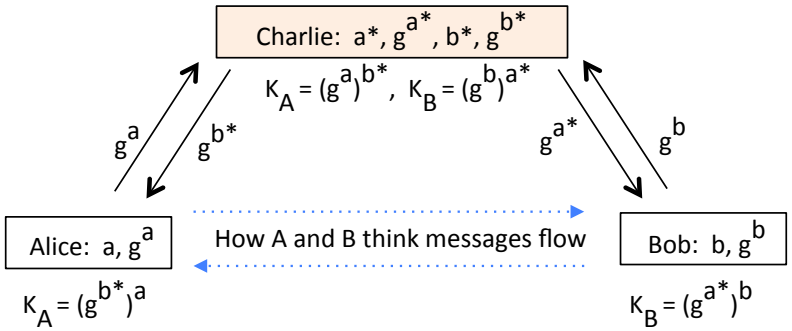



Figure 4.4: Middle-person attack on unauthenticated Diffie-Hellman key agreement. The normal DH key computed by both $A$ and $B$ would be $g^{ab}$. After key agreement, $C$ can use $K_A$ and $K_B$ to decrypt and re-encrypt messages that $A$ and $B$ send intended for each other.

you see the pattern now: an active attacker may replace exponentials with others that generate small subgroups, forcing $K$ into a small set easily searched. Such *small-subgroup attacks* (also called *subgroup confinement attacks*) are discussed in detail in Section 4.8; extra protocol checks to rule out these cases are easy, but essential.

**BASIC DIFFIE-HELLMAN IS UNAUTHENTICATED.** The basic DH protocol above is secure against *passive attack* (i.e., eavesdroppers), but protection is needed against *active attackers* who may inject or alter messages—such as in the small-subgroup attack just noted. We now discuss a second active attack, possible because neither $A$ nor $B$ knows the identity of the party it shares $K$ with, and thus that party might be...an adversary! This *middle-person attack* requires a defense other than simple tests on exchanged data.

**MIDDLE-PERSON ATTACK.** We first describe the classic *middle-person attack*, also called *man-in-the-middle* (MITM), on unauthenticated Diffie-Hellman (Fig. 4.4); we then discuss it generally. Legitimate parties $A$ and $B$ wish to carry out standard DH as above, with parameters $g, p$ and private values $a$, $b$. A sends $g^a$ intended for $B$. Attacker C (Charlie) creates private values $a^*$, $b^*$, and public exponentials $g^{a^*}$, $g^{b^*}$. $C$ intercepts and replaces $g^a$, sending to $B$ instead $g^{a^*}$. $B$ replies with $g^b$, which $C$ intercepts, sending instead $g^{b^*}$ to $A$. $A$ computes session key $K_A = g^{b^* \cdot a}$, while $B$ computes $K_B = g^{a^* \cdot b}$; these differ. Neither Alice nor Bob has actually communicated with the other, but from a protocol viewpoint, $C$ has carried out one "legitimate" key agreement with $A$, and another with $B$, and can compute both $K_A$ and $K_B$. Now any subsequent messages $A$ sends for $B$ (encrypted under $K_A$), can be decrypted by $C$, and re-encrypted under $K_B$ before forwarding on to $B$; analogously for messages encrypted (under $K_B$) by $B$ for $A$. In the view of both $A$ and $B$, all is well—their key agreement seemed fine, and encryption/decryption also works fine.

$C$ may now read all information as it goes by, alter any messages at will before forwarding, or inject new messages. Independent of DH, middle-person type attacks are a general threat—e.g., when a browser connects to a web site, if regular HTTP is used (i.e., unsecured), there is a risk that information flow is proxied through an intermediate site before proceeding to the final destination. Rich networking functionality and protocols, designed for legitimate purposes including testing and debugging, typically make

this quite easy. The next chess example is related.

**Example** *(Grandmaster postal chess).* A chess novice, Charlie, seeking to dishonestly raise his international chess rating, does so by engaging two grandmasters as follows. Charlie offers to play black (second move) against Alexis, and white against Boris. Alexis and Boris don't know there are two games going on, because naturally, games are conducted over the Internet (a few are still played by postal mail). Alexis makes the first move. Charlie plays that as his opening move against Boris (second game). Boris responds, and Charlie uses that move as his first-move response in the first game. And so on. In essence, the grandmasters are playing against each other, and Charlie is relaying moves. They probably draw (the most common outcome among grandmasters), and both congratulate Charlie on having done so well. Charlie improves his rating as a result of two draws with top-ranked players; his rating also improves if one grandmaster wins, in which case Charlie records one win and one loss.

**PROVING KNOWLEDGE OF A SECRET DOES NOT RULE OUT MIDDLE-PERSONS.** Note that if you (Alice) believe you are talking to Bob, but Charlie is a middle-person between you, you can't detect that by asking your presumed-Bob correspondent to send you the Bob-to-Alice password—since middle-person Charlie, pretending to be you (Alice), could just request Bob to send that password to him (impersonating you, Alice), and then relay it on to you. (Does this sound like a *relay attack*, above?)

**STS PROTOCOL.** The *Station-to-Station (STS)* protocol turns unauthenticated DH into authenticated DH. Section 4.4 discusses the properties it provides. Whereas EKE (Section 4.5) relies on passwords for authentication, STS uses digital signatures. If RSA signatures are used, let $A$'s signature on message $m$ be $S_A(m) = (H(m))^d \bmod n$, where $d_A = (d,n)$ is $A$'s signing private key and $H$ is a hash algorithm such as SHA-3 with result $h$ truncated so that $h < n$. In STS here, all data other than the two exponentials is encrypted under the resulting key $K$. The set-up is as in DH above. $\{m\}_K$ here denotes symmetric encryption (e.g., AES) of $m$ with key $K$; and $S_A(x,y)$ is the signature (tag) resulting from the signature operation over the concatenation of $x$ and $y$.

(1) $A \to B$ : $g^a$ ... (mod $p$) reduction omitted for visual appeal
(2) $A \leftarrow B$ : $g^b,\ \{S_B(g^b,g^a),\text{cert}_B\}_K$ ... symmetric encryption with shared key $K$
(3) $A \to B$ : $\{S_A(g^a,g^b),\text{cert}_A\}_K$ ... $S_A(m)$ is $A$'s signature on $m$

The Diffie-Hellman key $K$ is computed by each party as in steps (1), (2) of basic DH above. The public-key certificates $\text{cert}_A$, $\text{cert}_B$ (Chapter 8) are not needed if each party has an authentic copy of the other's signature verification public key, in which case they can be replaced by identifiers; but included as here within the encrypted data, they remain hidden from a network eavesdropper (preserving anonymity, depending on other details).

Studying examples of attacks on flawed protocols as in earlier sections, and flawed DH protocols also in Section 4.5, helps develop our intuition, and puts us in a position to systematically consider properties that are necessary or desirable in authentication and key establishment protocols. The next section discusses properties these protocols provide.

## 4.4 Key authentication properties and goals

**PROTOCOL GOALS AND PROPERTIES.** Key establishment protocols arrange shared secret keys. Basic requirements on a session key are that it be fresh, sufficiently long, random, and that parties know whom it is shared with. These define what may be called a *good key*. By *fresh* we mean a value that is new (not re-used from a previous session).

**FORWARD SECRECY.** Preferably, session keys also have forward secrecy. A protocol provides *forward secrecy* if disclosure of long-term secret keys does not compromise the secrecy of session keys from earlier runs. If old session keys cannot be reconstructed from long-term keys alone, and session keying material itself is *ephemeral*, i.e., vanishes at the end of the session,[4] then the secrecy of communications in a current session will remain secret into the future (even if other keys are compromised later). Diffie-Hellman agreement provides this property if for each protocol run both: (i) secrets ($a$, $b$) are fresh, making resulting secret $K$ fresh; and (ii) after the session, these secrets are securely deleted (i.e., unrecoverably removed from all memory storage they occupied).

**KNOWN-KEY SECURITY.** Forward secrecy means compromise of long-term keys can't expose previous session keys. A different concern is the impact of compromised session keys. A key establishment protocol has *known-key security* if compromised session keys do not put at risk future key management—compromised session keys should not allow later impersonation, or compromise of future session keys. We certainly never expect long-term keys to be at risk by compromise of session keys (the basic role of long-term keys is to play a part in establishing short-term keys, not the opposite).

**ENTITY AUTHENTICATION, LIVENESS, KEY-USE CONFIRMATION.** If a session key is for use in a real-time communication, key establishment and entity authentication should be done jointly in one integrated protocol (cf. Section 4.1). *Entity authentication* provides assurance that an identified far-end party is involved in the protocol (actually participating, active at the present instant); it thus provides a *liveness* property, not available in store-and-forward communication like email. A different property, possible in real time or store-and-forward, is *key-use confirmation*: one party having explicit evidence that another has a correct session or data key, via received data demonstrating knowledge of the correct key. Key-use confirmation may come without a known identity—for example, as provided by unauthenticated DH followed by key-use on known data.

**IMPLICIT AUTHENTICATION, EXPLICIT AUTHENTICATION.** If RSA encryption is used for key transfer from Alice to Bob, then Alice knows that Bob is the only party that *can* recover the key—but she does not know whether Bob has actually received it. This is an example of *implicit key authentication*—key establishment whereby the scope of who possibly has key access is narrowed to a specifically identified party, but possession is not confirmed. If Alice has such implicit key-auth, and then receives key-use confirmation, this could have originated from no one but Bob—and is thus explicit evidence that Bob has the key. Implicit key-auth plus key-use confirmation defines *explicit key authentication* (Fig. 4.5). Note that all three of these are properties that a protocol delivers to either one

[4] Preventing later reconstruction is a motivating example for principle P16 (REMNANT-REMOVAL).

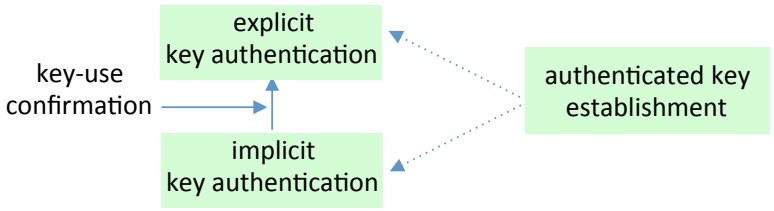



Figure 4.5: Key-authentication terminology and properties. Explicit and implicit key authentication are both authenticated key establishment (with, and without, key-use confirmation). Key establishment protocols may or may not provide *entity authentication*.

or both parties, based on the messages received and the information stored locally.

‡**STS AUTHENTICATION PROPERTIES.** In STS above, $A$ receives the encrypted message (2), decrypts, and verifies $B$'s digital signature on the two exponentials, checking that these are the properly ordered pair agreeing with that sent in (1). Verification success provides key-use confirmation to $A$. (We reason from the viewpoint of $A$; analogous reasoning provides these properties to $B$. The reasoning is informal, but gives a sense of how properties might be established rigorously.) The signature in (2) is over a fresh value $A$ just sent in (1); the fresh values actually play dual roles of DH exponentials and random-number TVPs. $B$'s signature over a fresh value assures $A$ that $B$ is involved in real time. Anyone can sign the pair of exponentials sent cleartext in (1) and (2), so the signature alone doesn't provide implicit key authentication; but the signature is encrypted by the fresh session key $K$, only a party having chosen one of the two DH private keys can compute $K$, and we reason that the far-end party knowing $K$ is the same one that did the signing. In essence, $B$'s signature on the exponentials now delivers implicit key authentication. The earlier-reasoned key-use confirmation combined with this provides explicit key authentication. Overall, STS provides to both parties: key agreement, entity authentication (not fully reasoned here), and explicit key authentication.

‡**Exercise** (BAN logic). The Burrows-Abadi-Needham *logic of authentication* is a systematic method for manually studying authentication and authenticated key establishment protocols and reasoning about their properties and the goals achieved.
(a) Summarize the main steps involved in a BAN logic proof (hint: [12]).
(b) What did BAN analysis of the X.509 authentication protocol find? ([17]; [33, p.510])
(c) Summarize the ideas used to add reasoning about public-key agreement to BAN [44].
(d) Summarize the variety of beliefs that parties in authenticated key establishment protocol may have about keys (hint: [9, Ch.2], and for background [33, Ch.12]).

## 4.5 Password-authenticated key exchange: EKE and SPEKE

*Password-authenticated key exchange* (PAKE) protocols use passwords to establish authenticated session keys. Since passwords are typically *weak secrets* (page 95 and below), these protocols must be carefully designed to resist offline password-guessing attacks. We discuss EKE (encrypted key exchange), building up to DH-EKE through several variants,

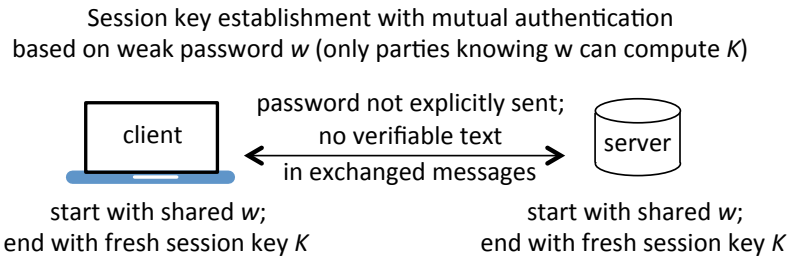



Figure 4.6: Password-authenticated key exchange (PAKE). If the password is user-entered when the protocol starts, and removed from memory after the protocol, then there is no client-side requirement for secure storage of long-term secrets.

and then briefly an alternative called SPEKE. The objective is not only to present the final protocols, but to gain intuition along the way, including about types of attacks to beware of—and to deliver the message that protocol design is tricky even for experts.[5]

**PAKE GOALS AND MOTIVATION.** Password-based protocols often convert user-chosen passwords into symmetric keys using a *key derivation function* (KDF, Section 3.2). As noted there, unless care is taken, clever attacks may discover passwords because:

1. protocol data visible to an attacker may allow testing individual password guesses (if it can serve as *verifiable text* analogous to hashes in a stolen password file); and
2. user-chosen passwords are *weak secrets*—a large proportion fall into small and predictable subsets of the full password space (recall Figure 3.2), or sometimes the full space itself is so small that its entirety can be searched; thus a correct guess is often expected within the number of guesses that an attacker is able to execute.

EKE and other PAKE protocols are key establishment protocols that use passwords as the basis for *mutual authentication* of established keys, without explicitly revealing the password to the far end, and aim to resist *offline* guessing attacks even if the passwords are user-chosen (weak). The protocol design must thus assure, among other things, that the protocol data sent "on the wire" contains no verifiable text as noted.

**NAIVE KEY MANAGEMENT EXAMPLE.** To build towards an understanding of EKE, we first see how simpler protocols fail. Let's explore how an attack may succeed. Let $w$ be a weak password, and $W = f(w)$ be a symmetric key (e.g., a 128-bit key for AES encryption, derived from $w$ by SHA-3 or any suitable key derivation function). Alice and Bob share (secret, long-term) key $W$; as earlier, "w" and "W" are mnemonic reminders of a weak secret from a user-chosen password. Now for this communication session, Alice generates a random $r$-bit (symmetric) session key $K$, encrypts it with $W$, and sends:

(1) $A \rightarrow B:\ C_1 = \{K\}_W$ ... i.e., $C_1$ is the symmetric encryption of $K$ under key $W$

(2) $A \leftarrow B:\ C_2 = \{T\}_K$ ... where text $T$ is a secret formula to be protected

On receiving (1), Bob decrypts with $W$; they then share session key $K$ to encrypt $T$. But an attacker could intercept and record $C_1, C_2$, then see whether the following "attack" succeeds, where $\{m\}_{K^{-1}}$ denotes decryption of $m$ with symmetric key $K$:

[5] This is a specific instance of principle P9 (TIME-TESTED-TOOLS).

1. Do a trial decryption of $C_1 = \{K\}_W$ with candidate key $W^*$.
   Call the resulting candidate session key $K^* = \{\{K\}_W\}_{W^{*-1}}$
2. Use $K^*$ to do a trial decryption of $C_2$, calling the result $T^*$.
3. Examine $T^*$ for recognizable redundancy (e.g., English text, ASCII coded).[6]
   If recognizable, conclude that the candidate key ($W^*$) is correct (i.e., the real $W$).

**VERIFIABLE TEXT.** In the above protocol, it is the *verifiable text* that puts the long-term weak secret $W$ at risk. The attack is *passive* (data is observed but no message flows are altered), and *offline* as discussed below. Will the attack succeed in recovering $w$?
Case 1 ($w$ is a long and random string, e.g., 80 to 128 bits): unlikely.
Case 2 ($w$ is a user-chosen password, often guessable in under $2^{20}$ or $2^{30}$ tries): likely.
In practice, "dictionaries" of 100,000 words (and variations) are wildly successful.

**PASSWORD-GUESSING STRATEGY TERMINOLOGY.** A *dictionary attack* (see also Chapter 3) refers to methods that typically step through a list of password candidates $w^*$ ordered by highest probability or expected chance of success. Here our interest is *offline* such methods (with no per-guess interaction with a legitimate verifier—thus incorrect guesses go undetected). Sufficiently short lists are processed fully; on longer lists, such an attack often achieves "early success", and if not may quit and move on to different targets. Such lists may be explicitly stored or dynamically enumerated. The method may use pre-computed tables (e.g., of password hashes per Chapter 3), partial tables (as in *rainbow tables* with time-memory tradeoff), or no tables. *Online guessing*, a second main password attack approach, requires per-guess interaction with a legitimate server. Note that the unqualified terms *offline guessing*, *guessing attack*, *brute-force guessing* and *exhaustive attack* often cause unnecessary ambiguity, unless explained further.

**GENERIC-NOTATION EKE.** With the above motivation, consider now the following conceptual protocol. A (Alice) and B (Bob) again pre-share a symmetric key $W$ derived from a password $w$. $\{x\}_W$ will mean symmetric encryption of $x$ under key $W$, and $E_{e_A}(x)$ denotes asymmetric (think: RSA) encryption of $x$ using $A$'s public key $e_A$.

(1) $A \rightarrow B$ : $A$, $\{e_A\}_W$ ... identifier $A$ signals whose shared secret to use
(2) $A \leftarrow B$ : $\{E_{e_A}(K)\}_W$ ... for convenience, define $C = E_{e_A}(K)$
(3) $A \rightarrow B$ : $\{T\}_K$ ... now use the session key to encrypt data $T$

Here, Alice generates a fresh (public, private) key pair $(e_A, d_A)$ then sends (1). Bob decrypts $\{e_A\}_W$ using $W$, recovering $e_A$; generates random symmetric key $K$; asymmetrically encrypts $K$ using $E_{e_A}$, super-encrypts with $W$, then sends (2). Alice decrypts (2) using first $W$ then $d_A$, to recover $K$. Now both Alice and Bob know $K$, a new session key.

**ANALYSIS.** What could an eavesdropper deduce on observing $\{e_A\}_W$, $\{E_{e_A}(K)\}_W$, $\{T\}_K$? To test a candidate password guess $W^*$, he could use $W^*$ to decrypt $\{e_A\}_W$, producing candidate public key ${e_A}^*$. Could he then easily verify whether ${e_A}^*$ is the public key in use? Let's see. Using the same guess $W^*$, he can trial-decrypt $\{E_{e_A}(K)\}_W$ to get candidate $C^* = E_{e_A}(K)^*$; but this *appears* to be of little help unless he can easily test whether candidate $C^*$ is the correct $C$. Since $K$ is also unknown, it seems the attack fails, as he must at the same time (for each $W^*$) guess candidates $K^*$ to find $K$ such that $\{\{T\}_K\}_{K^{*-1}}$

[6] ASCII's 7-bit code, commonly stored in 8-bits with top bit 0, enables an ideal test in such an attack.

is recognizable as a correct result. Alternatively: trying to deduce $K$ from candidate $C^* = E_{e_A}(K)^*$ is equivalent to defeating the public-key scheme (which we assume is not possible).

So *if* $e_A$ is *truly* random, guesses for $W^*$ can't be confirmed.[7] Apparently no "check-word" is available to test success. The attacker faces simultaneous unknowns $W$, $K$; even if the password is from a small predictable space, neither spaces of $K$ nor $e_A$ are small. (Aside: an attacker could always get "lucky" with a single online guess, which would be verified by protocol success—but EKE aims to stop offline guessing of weak passwords.)

**TIGHTENING AUTHENTICATION.** We now add messages to provide entity authentication including key-use confirmation of session key $K$. The time-variant parameters, as part of this, chain protocol messages, addressing common attacks noted in Section 4.2.

(1) $A \rightarrow B:\ A, \{e_A\}_W$
(2) $A \leftarrow B: \{E_{e_A}(K)\}_W, \{r_B\}_K$ ... $r_B$ is $B$'s random number for challenge-response
(3) $A \rightarrow B: \{r_A, r_B\}_K$ ... B recovers $r_A$ and $r_B$, aborts if mismatches earlier $r_B$
(4) $A \leftarrow B: \{r_A\}_K$ ... $A$ checks $r_A$ matches; if yes, accepts $K$ as session key

**RSA-EKE (FALSE START).** The above conceptual version is appealing. Can we instantiate it using RSA (Chapter 2) as the public-key system? This turns out to be tricky. From (1), a guessed password $W^*$ allows deduction of a corresponding candidate $e_A^*$; we must ensure that no verifiable text allows an attacker to easily test $e_A^*$ for "validity". If the RSA public key is $e_A = (e,n)$, how should these be encrypted in (1) so that such an $e_A^*$ recovered using an incorrect password guess appears no different than a random bitstring? This is problematic due to both $e$ and $n$. Regarding $e$: valid RSA exponents are odd, but half of trial decryptions $e_A^*$ will yield an even $e^*$. An attacker could eliminate half the candidates from a dictionary, *and do likewise for each protocol run* if fresh RSA keys are used in each. We call data such as $e$ a *partitioning text* (explained below). This particular case can be addressed by randomly (for 50% of protocol runs) adding 1 to the RSA exponent (the receiving party simply drops the bit if present, knowing the process).

Regarding $n$: an attacker can attempt to factor a recovered $n^*$. A genuine RSA modulus $n$ has the uncommon property of being a product of two large primes; most numbers of similar size have many small prime factors that well-known methods find quickly—signaling an invalid RSA modulus, thus an invalid guess $W^*$. This allows a similar partitioning attack. Rather than pursue these and other RSA-EKE implementation challenges further, we instead move on to the more promising DH-EKE—having learned the lesson that in security implementations, there is almost always a devil in the details.

**PARTITION ATTACKS.** Before moving on, we explain the general idea of a *partition attack*. Suppose a particular data field (*partitioning text*) in a protocol message leaks information allowing a partition of a candidate-password list (dictionary) into two disjoint sets: those still possible, and those eliminated. If the partition is roughly equal size, a dictionary of $2^u$ entries is cut to $2^{u-1}$. If each protocol run randomizes data such that dictionary candidates are randomly redistributed, then observing a second protocol run

[7] You are being led into a trap; a public key randomly and properly selected will not necessarily be the same as a random string (but read on).

reduces the dictionary again by half to $2^{u-2}$, and so on logarithmically. How does this compare to verifiable text, such as a password hash? If searching for a weak secret using a dictionary list $D$, one verifiable text may allow an offline search through $D$, finding the weak secret if it is among the entries in $D$. In contrast, a partition attack collects test data from numerous protocol runs, and each narrows a dictionary by some fraction; each involves the same weak secret, randomized differently, to the attacker's benefit. This attack strategy re-appears regularly in crypto-security. We say that each protocol run *leaks* information about the secret being sought.

**DH-EKE.** As in basic Diffie-Hellman (Section 4.3), we need parameters $g, p$. A first question is whether it is safe to transmit $\{p\}_W$ in a protocol message. Answer: no. Testing a candidate $p^*$ for primality is easy in practice, even for very large $p$, so transmitting $\{p\}_W$ would introduce verifiable text against which to test candidate guesses for $W$. So assume a fixed and known DH prime $p$ and generator $g$ for the multiplicative group of integers mod $p$.[8] To add authentication to basic DH (being unauthenticated, it is vulnerable to *middle-person attacks*), the exponentials are encrypted with pre-shared key $W$:

(1) $A \to B$ : $A, \{g^a\}_W$ ... the key agreement public key $g^a$ is short for $g^a \pmod p$
(2) $A \leftarrow B$ : $\{g^b\}_W$

Each party uses $W$ to recover regular DH exponentials, before executing DH key agreement. Note that DH private keys $a$ and $b$ (unlike an RSA modulus) have no predictable form, being simply non-zero random numbers of sufficient length. The idea is that a middle-person attack is no longer possible because:

(i) the attacker, not knowing $W$, cannot recover the DH exponentials; and

(ii) since $a$ is random, we hope (see Note 1 below) that $g^a$ is also random and leaks no information to guesses $W^*$ for $W$. We now have a full illustrative version of DH-EKE:

(1) $A \to B$ : $A, \{g^a\}_W$ ... symmetrically encrypt $g^a$ under key $W$
(2) $A \leftarrow B$ : $\{g^b\}_W, \{r_B\}_K$ ... $r_B$ is B's random challenge
(3) $A \to B$ : $\{r_A, r_B\}_K$ ... $B$ checks that $r_B$ matches earlier
(4) $A \leftarrow B$ : $\{r_A\}_K$ ... $A$ checks that $r_A$ matches earlier

Distinct from the conceptual EKE (above), here each party computes fresh session key $K$ from the result of DH key agreement, rather than $B$ generating and transmitting it to $A$. This provides *forward secrecy* (Section 4.4). The protocol can be viewed in practical terms as using passwords to encrypt DH exponentials, or abstractly as using a shared weak secret to symmetrically encrypt ephemeral public keys that are essentially random strings.

**NOTE 1.** The above hope is false. If modulus $p$ has bitlength $n$, then valid exponentials $x$ (and $y$) will satisfy $x < p < 2^n$; any candidate $W^*$ that results in a trial-decrypted $x$ (or $y$) in the range $p \le x < 2^n$ is verifiably wrong. Thus each observation of a $W$-encrypted exponential *partitions* the list of dictionary candidates into two disjoint sets, one of which can be discarded. The fraction of remaining candidates that remain contenders, i.e., the fraction yielding a result less than $p$, is $p/2^n < 1$; thus if $t$ protocol runs are observed, each yielding two exponentials to test on, the fraction of a dictionary that remains eligible is $(p/2^n)^{2t}$. This offline attack is ameliorated by choosing $p$ as close to $2^n$ as practical; an

[8] Here $p = Rq + 1$ for a suitably large prime $q$, and thus $p$ is *PH-safe* per Section 4.8.

alternate amelioration is also suggested in the original EKE paper (Section 4.9).

**SPEKE.** An elegant alternative, SPEKE (simple password exponential key exchange) addresses the same problem as DH-EKE, but combines DH exponentials with passwords without using symmetric encryption in the key agreement itself. The final steps for key-use confirmation can be done as in DH-EKE, i.e., the symmetric encryptions in steps (2)-(4) above. For notation here, let $w$ denote the weak secret (password).[9]

(1) $A \rightarrow B:\ A, (w^{(p-1)/q})^a$ ... this is just $f(w)^a$ if we write $f(w) = w^{(p-1)/q}$
(2) $A \leftarrow B:\ (w^{(p-1)/q})^b$ ... and $f(w)^b$

Again exponentiation is mod $p$ (for $p = Rq+1$, $q$ a large prime). As before, $A$ and $B$ each raise the received value to the power of their own private value; now $K = w^{ab(p-1)/q}$.

Notes: If $R = 2$, then $(p-1)/q = 2$ and the exponentials are $w^{2a}$ and $w^{2b}$; such a $p$ is called a *safe prime* (Section 4.8). We can assume the base $w^{(p-1)/q}$ has order $q$ (which as noted, is large).[10] The order of the base bounds the number of resulting values, and small-order bases must be avoided as with basic DH—recall the *small-subgroup attack*. Because an active attacker might manipulate the exchanged exponentials to carry out such an attack, before proceeding to use key $K$, $A$ and $B$ must implement tests as follows.

- Case: $p$ is a *safe prime*. Check that: $K \neq 0$, 1, or $p-1 \pmod p$.
- Otherwise: do the above check, plus confirm that: $x^q = 1 \pmod p$. This confirms $x$ is in the group $G_q$. Here $x$ denotes the received exponential in (1), (2) respectively.

‡**Example** *(Flawed SPEKE).* One of SPEKE's two originally proposed versions had a serious flaw. We explain it here, using a key-use confirmation design yielding a minimal three-message protocol originally proposed for EKE, but adopted by SPEKE.

(1) $A \rightarrow B:\ A, g^{wa}$ ... this is $f(w)^a$ for $f(w) = g^w$; $g$ is chosen to have order $q$
(2) $A \leftarrow B:\ g^{wb}, \{g^{wb}\}_K$ ... $B$'s exponential doubles as a random number
(3) $A \rightarrow B:\ \{H(g^{wb})\}_K$ ... key-use confirmation in (2) and (3)

This version of SPEKE exchanges $f(w)^a$ and $f(w)^b$ where $f(w) = g^w$ and $g = g_q$ generates a subgroup of order $q$ (found per Section 4.8). $A$ sends $g^{wa}$ with resulting $K = g^{wab}$. For a weak secret $w$, this version falls to a dictionary attack after an attacker C (Charlie) first initiates a single (failed) protocol run, as follows. After $A$ sends $g^{wa}$, $C$ responds with $g^x$ (not $g^{wb}$) for a random $x$—he need not follow the protocol! $A$ will compute $K = (g^x)^a = g^{xa}$. $C$ receives $g^{wa}$, and knowing $x$, can compute $(g^{wa})^x$; he can also make offline guesses of $w^*$, and knowing $q$, computes the (mod $q$) inverse of $w^*$ by solving $z \cdot w^* \equiv 1 \pmod q$ for $z = (w^*)^{-1}$. Now for each guess $w^*$ he computes $K^* = g^{wax(w^{*-1})}$; the key point is that for a correct guess $w^*$, $K^*$ will equal $g^{ax}$, which is $A$'s version of $K$. Using $A$'s key-use confirmation in (3), $C$ independently computes that value using $K^*$ (possible because $C$ also knows the value $K$ is being confirmed on), and a match confirms $w = w^*$; otherwise $C$ moves on to test the next guess for $w^*$, until success. This attack exploits two errors: failure to anticipate combining a dictionary attack with a one-session active attack, and a key-use confirmation design that provides verifying text.

[9] When $w$ is first processed by a hash function, we use $W = H(w)$; a different SPEKE variation does so.

[10] By F7 (Section 4.8), the order is $q$ or 1 (exponentiating by $R = (p-1)/q$ forces it into the order-$q$ group). For it to be 1, $q$ must divide $R$; this can be avoided by choice of $p$, and is ruled out by later checking: $K \neq 1$.

‡**Exercise** (SRP, OKE, J-PAKE). Summarize the technical details of the following password-authenticated key exchange alternatives to EKE and SPEKE.
(a) SRP/Secure Remote Password (hint: [49, 48]).
(b) OKE/Open Key Exchange (hint: [30], but also [31] or [9, Chapter 7]).
(c) J-PAKE/PAKE by Juggling (hint: [21, 22, 20]).

## 4.6 ‡Weak secrets and forward search in authentication

A *weak secret*, derived from a user-chosen password or short numeric PIN, can be found by an attacker in a feasible amount of time given appropriate circumstances—meaning, such circumstances must be precluded by design, if protocols are to be used with weak secrets in place of strong keys. EKE, SPEKE, and some other authenticated key establishment protocols do attempt to accommodate weak secrets—thus their protocol designs aim to preclude dictionary-type attacks.[11] Here we consider some further protocol examples and related *forward search attacks* that must be addressed when using public-key systems with weak secrets. A first example shows that verifiable text need not originate from a password hash file or a key agreement protocol.

**Example** *(Gong's SunOS-4.0 public-key system example).* User $A$ has encryption public-private key pair $(e_A, d_A)$ with the private key stored AES-encrypted under a key $W = f(w)$ derived from password $w$, denoted $\{d_A\}_W$. On login, the system decrypts using $W$ to recover $d_A$. Its correctness is verified by testing that the recovered $d_A$ (call it $d_A^*$) is indeed the inverse of $e_A$ as follows: choose a random $x$, compute $E_{e_A}(x)$, and decrypt with $d_A^*$ to see whether $x$ returns. An attacker can similarly test (offline) password guesses $W^*$ by computing ${d_A}^* = \{\{d_A\}_W\}_{W^{*-1}}$, choosing any value $x$, and testing whether $x = D_{d_A^*}(E_{e_A}(x))$. If yes, then very likely $W^* = W$ and ${d_A}^* = d_A$; false positives are eliminated by trying other $x$. Thus despite $d_A$ being a strong cryptographic key, the weak secret makes it vulnerable to discovery. The stored values $e_A$ and $\{d_A\}_W$ serve as *verifiable text*.

RECOGNIZABLE FORMATS. The general problem is that testing is possible as a result of recognizable formats. Examples are: a timestamp field for which the recovered value is a plausible timestamp; a field having a verifiable structure (e.g., a URL with dots and a known top-level domain/TLD); a checksum used for message integrity; a known text field (e.g., fixed identifier, server name, service name). For example, with $K$ denoting a symmetric key, suppose an attacker observes $\{(m,n)\}_K$ and that $m$ is *known plaintext*. Guess a $K^*$ for $K$, compute $\{\{m,n\}_K\}_{K^{*-1}}$ and test whether $m$ is found therein. Yes means $K^*$ is probably correct; no means definitely wrong.

FORWARD-SEARCH ATTACKS. By definition, public-key systems have publicly available encryption keys. This raises a vulnerability: directly encrypting a weak secret with a public-key system produces verifiable text. Let $E_{e_A}(x)$ denote public-key encryption of $x$, a value from a small space $S$ (small number of elements). An attacker can carry out the following *forward search attack*. Compute $E_{e_A}(x)$ for every element $x$ in $S$. Any value $C = E_{e_A}(w)$ later produced, e.g., in a protocol, can be compared to such values. This

[11] Failure to do so would violate principle P12 (SUFFICIENT-WORK-FACTOR).

attack is likewise possible if by guessing values $w^*$, an attacker can derive from protocol data, values serving as verifiable text to compare to forward-search values. Although $E_{e_A}(x)$ encrypts $x$, the public-key property means it is publicly computable like a one-way hash; thus the similarity to dictionary attacks on protocol data or password hashes.

A standard defense is to choose a sufficiently long random number $r$, and compute $E_{e_A}(r,x)$. The intended recipient recovers $r$ and $x$ (simply throwing away the $r$, which has served its purpose). A value $r = c$ used in this manner is sometimes called a *confounder* in other contexts, as it confuses or precludes attacks. Note the analogy to password *salting*. More generally, attacks against weak secrets are often stopped by "randomizing" protocol data related to weak secrets, in the sense of removing redundancies, expected values, and recognizable formats or structural properties that may otherwise be exploited.

**Example** *(Weak secrets in challenge-response authentication).* Consider this protocol to prove knowledge of a weak secret $W$; $r$ is a random number; $f$ is a known function.

(1) $A \rightarrow B$ : $\{r\}_W$ ... unilateral authentication of $B$ to $A$
(2) $A \leftarrow B$ : $\{f(r)\}_W$ ... use simple $f(r) \neq r$ to prevent reflection attack

Neither (1) nor (2) contains verifiable text alone as $r$ and $f(r)$ are random, but jointly they can be attacked: guess $W^*$ for $W$, decrypt (1) to recover $r^*$, compute $f(r^*)$, test equality with (2) decrypted using $W^*$. The attack is stopped by using two unrelated keys:

(1′) $A \rightarrow B$ : $\{r\}_{K_1}$
(2′) $A \leftarrow B$ : $\{f(r)\}_{K_2}$

In this case, for fixed $r$, for *any* guessed $K_1$, we expect a $K_2$ exists such that $\{r\}_{K_1} = \{f(r)\}_{K_2}$ so an attacker cannot easily confirm correct guesses. Rather than ask users to remember two distinct passwords (yielding $K_1 = W_1, K_2 = W$), consider these changes. Choose a public-private key pair (for $B$). The public key replaces the functionality of $K_1$; the private key stays on $B$'s computer. A sufficiently large random number $c_A$ is also used as a *confounder* to preclude a *forward search attack*. To illustrate confounders, we artificially constrain $f$ to the trivially-inverted function $f(r) = r + 1$ (although our present problem is more easily solved by a one-way hash function[12]):

(1) $A \rightarrow B : E_{K_1}(c_A, r)$ ... $K_1$ is the encryption public key of user $B$; $A$ is the server
(2) $A \leftarrow B : \{f(r)\}_W$ ... $B$ proves knowledge of password-derived key $W = K_2$

This stops the guessing attack, since to recover $r$ from (1) for a test, an attacker would need to guess either (i) $W$ as well as the private key related to $K_1$; or (ii) both $W$ and $c_A$.

**Exercise** (Protocol analysis). These questions relate to the example above.
(a) List the precise steps in attack alternatives (i) and (ii), and with concrete examples of plausible search space sizes, confirm that the attacks take more than a feasible time.
(b) Explain, specifying precise attack steps, how an active attacker can still learn $W$.
Hint: consider an attacker $A'$ sending the first message to $B$, and note that $A'$ knows $r$.

**Example** *(Forward search: authentication-only protocol).* As a final example of disrupting forward search, this unilateral authentication protocol proves that user $A$ knows a weak secret $W_A = H(w)$ computed on user-entry of $A$'s numeric PIN $w$. $A$'s device contains no keying material other than the bank's public key $e_S$.

[12] This is noted and discussed further by Gong [19].

(1) $A \leftarrow S:\ r_S$ ... random challenge from server $S$
(2) $A \rightarrow S:\ E_{e_S}\{A, c_A, \{r_S\}_{W_A}\}$ ... confounder $c_A$; encryption public key $e_S$ of $S$

Receiving (2), $S$ decrypts, uses identifier $A$ to retrieve the shared secret $W_A$, and uses that to recover $r_S$ to compare to the challenge in (1). If $c_A$ is removed, an eavesdropper knowing $A$ can compute $E_{e_S}\{A, \{r_S\}_{W_A^*}\}$ for guesses $W_A^*$, and compare to the value seen in (2).

## 4.7 ‡Single sign-on (SSO) and federated identity systems

A *single sign-on* (SSO) system is an architecture allowing a user to authenticate once at the beginning of a session or shift, and then without re-authenticating separately for each, access a set of services or resources from *service providers* (SPs, also called *relying parties*, RPs). Thus users manage only one credential, e.g., a master userid-password pair or hardware token. The process involves third parties called *identity providers* (IdPs), which access credentials or create *authenticators* (data tokens) from the initial authentication for later identity representations. Administratively, SSO systems allow centralized account provisioning, management, and retraction of access privileges on employee termination. A downside is concentrated risk:[13] a single attack compromises many resources, and loss of a master credential denies access to many resources. In a *single credential system*, users similarly have one master credential, but must explicitly log in to each service with it.

**TYPES OF SSO SYSTEMS.** Different types of SSO systems are classified based on design architecture and application scenarios. We note three popular categories.

1. *Credential manager* (CM) systems including *password managers* (Chapter 3). These manage, on behalf of users, SP-specific credentials (passwords or crypto keys), usually encrypted under a key derived from the master credential (e.g., user password, or hardware token storing a crypto key). Individuals wishing to simplify their own password management may resort to password managers.
2. *Enterprise SSO* systems. These allow users to access resources controlled by a single administrative domain, and typically serve a *closed community* such as a corporate or government organization aiming to simplify life for employees. The Kerberos system (below) has long been popular in universities and Windows-based systems.
3. *Federated identity* (FI) systems. These support authentication across distinct administrative domains, and often serve non-enterprise or *open communities*, including web sites (SPs) seeking to lower registration barriers for new users. These are also called *web SSO* systems when user agents are browsers, although enterprise SSO systems also commonly use browser-based interfaces.

**FEDERATED IDENTITY SYSTEMS.** The design of FI systems specifies protocols for user identity registration, user-to-IdP authentication, IdP-to-SP authentication, and who controls the *name space* of user identities. An additional party, the *federation operator*, sets administrative and security policies; how these are enforced depends on the parties

[13] This is analogous to risks, and related principles, associated with password managers in Chapter 3.

involved. (In enterprise SSO systems, internal information technology staff and management are responsible.) Each user registers with an IdP, and on later requesting a service from an RP/SP, the user (browser) is redirected to authenticate to their IdP, and upon successful authentication, an IdP-created authentication token is conveyed to the RP (again by browser redirects). Thus IdPs must be recognized by (have security relationships with) the RPs their users wish to interact with, often in multiple administrative domains. User-to-IdP authentication may be by the user authenticating to a remote IdP over the web by suitable user authentication means, or to a local IdP hosted on their personal device.

**KERBEROS PROTOCOL (PASSWORD-BASED VERSION).** The simplified Kerberos protocol below provides mutual entity authentication and authenticated key establishment between a client $A$ and a server $B$ offering a service. It uses symmetric-key transport. A trusted KDC $T$ arranges key management. The name associates $A$, $B$ and $T$ with the three heads of the dog *Kerberos* in Greek mythology. $A$ and $B$ share no secrets *a priori*, but from a registration phase each shares with $T$ a symmetric key, denoted $k_{AT}$ and $k_{BT}$, typically password-derived. (Protocol security then relies in part on the properties of passwords.)

$A$ gets from $T$ a *ticket* of information encrypted for $B$ including an A-B session key $k_S$, $A$'s identity, and a lifetime $L$ (an end time constraining the ticket's validity); copies of $k_S$ and $L$ are also separately encrypted for $A$. The ticket and additional authenticator $auth_A$ sent by $A$ in (3), if they verify correctly, authenticate $A$ to $B$:

(1) $A \rightarrow T$: $A, B, n_A$ ... $n_A$ is a nonce from A
(2) $A \leftarrow T$: $tick_B, \{k_S, n_A, L, B\}_{k_{AT}}$ ... $tick_B = \{k_S, A, L\}_{k_{BT}}$ is for forwarding to $B$
(3) $A \rightarrow B$: $tick_B, auth_A$ ... $auth_A = \{A, t_A, k_A\}_{k_S}$, $t_A$ is time from $A$'s clock
(4) $A \leftarrow B$: $\{t_A, k_B\}_{k_S}$ ... $A$ checks that $t_A$ matches the value from (3)

On receiving (2), $A$ recovers parameters $k_S, n_A, L$ and $B$ (cross-checking that the identifiers $B$ in (1) and (2) match, that the nonces match, and saving $L$ for reference). The $auth_A$ received by $B$ in (3) reveals whether $A$ knows $k_S$; $B$ recovers parameters from $tick_B$ including $k_S$, uses it to recover the contents of $auth_A$, and checks that the identifiers match, that $t_A$ is valid per $B$'s timestamp policy, and that $B$'s local time is within lifetime $L$. Message (4) allows entity authentication of $B$ to $A$, as a matching $t_A$ indicates that $B$ knows $k_S$. The secondary keys $k_A$ and $k_B$ created by $A$ and $B$ are available for other purposes (e.g., a one-way function $f(k_A, k_B)$ may serve as a new session key independent of $k_S$). The use of timestamps creates the requirement of (secure) synchronized timeclocks.

**TICKET GRANTING TICKETS.** The above version gives the basic idea of Kerberos. In some Kerberos-based authentication services, before getting service tickets such as $tick_B$, an end-user $A$ must get an initial (time-limited) *ticket granting ticket* (TGT) token upon logging into a primary service account by password authentication or stronger means. The TGT token is used to obtain service tickets from $T$ replacing (1)-(2) above, by a protocol not detailed here. Such a system is more recognizable as an enterprise SSO system, but even in the version above, $T$ facilitates $A$ getting service from $B$ multiple times (within lifetime $L$), i.e., multiple iterations of (3)-(4) after (1)-(2) once.

## 4.8 ‡Cyclic groups and subgroup attacks on Diffie-Hellman

‡*This section may be skipped by those who did not enjoy mathematics in primary school.* The mathematical background herein is helpful for understanding the computations involved in, and security of, Diffie-Hellman key agreement (Section 4.3).

**MULTIPLICATIVE GROUPS AND GENERATORS.** If $p$ is a prime number, then the non-zero integers modulo $p$, i.e., the integers 1 to $p-1$, form a *multiplicative group*. The idea of a group is that combining two group elements using the group operation yields another group element (i.e., another integer in $[1, p-1]$); the group operation is multiplication followed by taking the remainder after dividing by $p$, which is what "mod $p$" means. If $p$ is prime, a *generator* $g$ always exists such that $g^i \pmod p$ for $i = 1, 2, ...$ generates all group elements, i.e., $g^1$, $g^2$, $g^3$, ..., $g^{p-1}$ mod $p$ is some re-ordering of the integers 1 to $p-1$. Note $g^p = g^1 \cdot g^{p-1}$, and $g^{p-1} = 1$ and the cycle begins again. Mathematicians, always eager to impress, use the fancy name $\mathcal{Z}_p^*$ for this algebraic structure; $\mathcal{Z}$ denotes integers, $p$ for prime, and asterisk for multiplicative group/zero removed. Don't worry—damage to young minds from exposure to this notation is likely short-term.

**Example** *(Multiplicative group of integers mod 11).* For $p = 11$ and $g = 2$, the integers 1 to $10 = p-1$ form a (cyclic) multiplicative group $\mathcal{Z}_{11}^*$. Computing $g^i \pmod p$, $i \geq 1$ yields $2, 4, 8, 5, 10, 9, 7, 3, 6, 1$. Thus $g = 2$ generates all 10 elements and is a generator.

**ORDER OF MULTIPLICATIVE GROUP, AND SUBGROUPS.** An alternate view of $\mathcal{Z}_p^*$ is as a *cyclic group* $G_n$; $G$ is for group, $n$ is the number of elements. Since $\mathcal{Z}_p^*$ is a multiplicative group with $p-1$ elements, we can view it abstractly as $G_{p-1}$. For $p$ prime, $G_{p-1}$ is cyclic. We define the *order* of each element $b$ to be the number of distinct elements its powers $b^i$ generate; or equivalently, the smallest positive $j$ such that $b^j \equiv 1 \pmod p$. A *generator* generates all elements in the group. Thus a generator for $G_t$ has order $t$. Every element of a cyclic group generates a cyclic *subgroup*—either the full group, or a smaller one. An example is given further below.

**GENERATORS, ORDERS OF ELEMENTS, OTHER FACTS.** We collect several further facts for use related to Diffie-Hellman security and subgroup attacks (Section 4.3). These can be formally established fairly easily; by playing around with small examples such as $Z_{11}^*$ with your morning coffee, you can confirm most of them—but our aim is to use them as tools, not to prove them. Below, $\phi(n)$ is the number of integers from 1 to $n$ whose greatest common divisor with $n$ is 1, e.g., $\phi(9) = 6$ as $i = 1, 2, 4, 5, 7, 8$ have $gcd(i, 9) = 1$.

These facts for cyclic groups $G_n$ apply to the cyclic group $\mathcal{Z}_p^* = G_{p-1}$. Commonly, $p = Rq + 1$ is the notation used for mod $p$ Diffie-Hellman, yielding a subgroup $G_q$ having $q$ elements ($q$ is a prime here). You may find it helpful to take your pencil and paper, and work through each of these facts F1–F7 using the example of Table 4.2 (page 116).

F1: In a cyclic group $G_n$, exponents can always be reduced modulo $n$. For $p = 11$ and generator $g = 2$: $2^{10} \pmod{11} \equiv 2^0 \pmod{11} \equiv 1$. Exponents reduce mod 10.

F2: i) All subgroups of a cyclic group are also cyclic.
ii) The order of a subgroup will divide the order of the group.
iii) The order of an element will divide the order of the group.

For $G_{p-1}$: the order of any element is either $d = 1$ or some $d$ that divides $p-1$.
For $G_q$: all elements thus have order either 1 or $q$ (since $q$ is prime).

F3: $G_n$ has exactly one subgroup of order $d$ for each positive divisor $d$ of its order $n$; and $\phi(d)$ elements of order $d$; and therefore $\phi(n)$ generators.

F4: If $g$ generates $G_n$, then $b = g^i$ is also a generator if and only if $gcd(i,n) = 1$.

F5: $b$ is a generator of $G_n$ if and only if, for each prime divisor $p_i$ of $n$: $b^{n/p_i} \neq 1$

F6: For generator $g$ of $G_n$, and any divisor $d$ of $n$: $h = g^{n/d}$ yields an order-$d$ element.

F7: Without knowing a generator for a cyclic group $G_n$, for any prime divisor $p_1$ of $n$, an element $b$ of order $p_1$ can be obtained as follows:

Select a random element $h$ and compute $b = h^{n/p_1}$; continue until $b \neq 1$.

(Obviously a $b = 1$ does not have prime order; and for $b \neq 1$, it lies in the unique subgroup of order $p_1$ and must itself be a generator, from F2 and F3.)

**Exercise** (Another $\mathcal{Z}_{11}^*$ generator). For $\mathcal{Z}_{11}^*$ as above, set $g = 3$. Does the sequence $g, g^2, g^3$... (all reduced mod 11) generate the full set, or just half? Find a generator other than $g = 2$, and list the sequence of elements it generates.

**Example** *(Subgroups of $\mathcal{Z}_{11}^*$).* Table 4.2 explores the subgroups of $\mathcal{Z}_{11}^*$, or $G_{10}$. We have seen that one generator for the full group is $g = 2$. The element $h = 3$ generates the order-5 cyclic subgroup $G_5$. The elements of $G_5$ can be represented as powers of $h$:

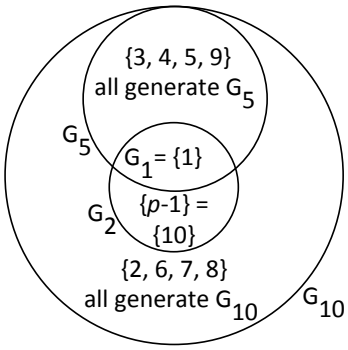
$$h^1 = 3,\ h^2 = 9,\ h^3 = 5,\ h^4 = 4,\ h^5 = 1 = h^0$$

To view $G_5$ in terms of the generator $g = 2$ of the full group, since $h = 3 = g^8$, this same ordered list $(3,9,5,4,1)$ represented in the form $(g^8)^i$ is:

$$(g^8)^1 = 3,\ (g^8)^2 = 9,\ (g^8)^3 = 5,\ (g^8)^4 = 4,\ (g^8)^5 = 1$$

Since for $p = 11$, exponents can be reduced mod 10, this is the same as: $g^8, g^6, g^4, g^2, g^0$. The divisors of 10 are 1, 2, 5 and 10; these are thus (by F2 above) the only possible orders

| Order | Subgroup | Generators |
|---|---|---|
| 1 | {1} | 1 |
| 2 | {1, 10} | 10 |
| 5 | {1, 3, 4, 5, 9} | 3, 4, 5, 9 |
| 10 | {1, 2, 3, ..., 10 } | 2, 6, 7, 8 |




| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | ⋯ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b = g^i$ | 2 | 4 | 8 | 5 | 10 | 9 | 7 | 3 | 6 | 1 | 2 | 4 | ⋯ |
| order of $b$ | 10 | 5 | 10 | 5 | 2 | 5 | 10 | 5 | 10 | 1 | 10 | 5 | ⋯ |

Table 4.2: The structure of subgroups of $\mathcal{Z}_{11}^* = G_{10}$. $G_{10}$ has four distinct subgroups. The lower table shows how elements can be represented as powers of the generator $g = 2$, and the orders of these elements. Note that $p = 11$ is a safe prime ($p - 1 = 2 \cdot 5$).

of subgroups, and also of elements in them. If you don't believe this, cross-check Table 4.2 with pencil and paper for yourself. (Yes, really!) Given any generator $g$ for $G_{10}$, it should be easy to see why $g^2$ is a generator for a subgroup half as large, and $g^5$ generates a subgroup one-fifth as large. At the right end of the lower table, the cycle repeats because here exponents are modulo 10, so $2^{10} \equiv 2^0 = 1 \bmod 11$ (as noted, for integers mod $p$, exponent arithmetic is modulo $p-1$). Note that element $b = 10$ has order 2 and is a member of both $G_2 = \{1,-1\}$ (the subgroup of order 2) and $G_{10}$. $b = 10 \equiv -1$ is not in the subgroup of order 5; and 2 does not divide 5. (Is that a coincidence? See F2.) Note that the indexes $i$ such that $gcd(i,10) = 1$ are $i = 1,3,7,9$, and these are the four values for which $g^i$ also generates the full group $G_{10}$. (Is this a coincidence? See F4.)

**Exercise** (Multiplicative groups: $p = 19,23,31$). By hand, replicate Table 4.2 for $Z_p^*$ for a) $p = 19 = 2 \cdot 3 \cdot 3 + 1$; b) $p = 23 = 2 \cdot 11 + 1$; and c) $p = 31 = 2 \cdot 3 \cdot 5 + 1$. (F2 will tell you what orders to expect; use F3 to confirm the number of generators found.)

**COMMENT ON EXPONENT ARITHMETIC (F1).** In $G_n$ notation we often think abstractly of "group operations" and associate elements with their exponent relative to a fixed generator, rather than their common implementation representation in integer arithmetic mod $p$. Consider $p = Rq + 1$. The multiplicative group $Z_p^*$ is a cyclic group of $p-1$ elements. In mod $p$ representation, "exponent arithmetic" can be done mod $p-1$ since that is the order of any generator. $Z_p^*$ has a cyclic subgroup $G_q$ of $q$ elements, and when expressing elements of $G_q$ as powers of a $G_q$ generator, exponent arithmetic is mod $q$. However, the subgroup operations are still implemented using mod $p$ (not mod $q$); the mod $q$ reduction is for dealing with exponents. Thus switching between $Z_p^*$ and $G_q$ notation, and between elements and their indexes (exponents), requires precision of thought. We mentally distinguish between implementation in "modular arithmetic", and "group operations". It may help to re-read this and work through an example with $q = 11$.

**SAFE PRIMES, DSA PRIMES, SECURE PRIMES.** Let $p$ be a prime modulus used for Diffie-Hellman (DH) exponentiation. The security of Diffie-Hellman key agreement relies on it being computationally difficult to compute discrete logarithms (Section 4.3). It turns out that the Pohlig-Hellman discrete log algorithm is quite efficient unless $p-1$ has a "large" prime factor $q$, where "large" means that $\sqrt{q}$ operations is beyond the computational power of an attacker. (So for example: if an attacker can carry out on the order of $2^t$ operations, $q$ should have bitlength more than $2t$; in practice, a base minimum for $t$ is 80 bits, while 128 bits offers a comfortable margin of safety.)

For Diffie-Hellman security, the following definitions are of use. Recall (F2 above) that the factors of $p-1$ determine the sizes of possible subgroups of $Z_p^* = G_{p-1}$.

1. A *PH-safe prime* is a prime $p$ such that $p-1$ itself has a "large" prime factor $q$ as described above. The motivation is the Pohlig-Hellman algorithm noted above. Larger $q$ causes no security harm (attacks become computationally more costly).
2. A *safe prime* is a prime $p = 2q+1$ where $q$ is also prime. $Z_p^*$ will then have order $p-1 = 2q$, and (by F2) will have exactly two proper cyclic subgroups: $G_q$ with $q$ elements, and $G_2$ with two elements $(1, p-1)$. Remember: $p-1 \equiv -1 \bmod p$.
3. A *DSA prime* is a prime $p = Rq+1$ with a large prime factor $q$ of $p-1$. Traditionally,

here $q$ is chosen large enough to be *PH-safe*, but not much larger. The idea is to facilitate DH computations in the resulting *DSA subgroup* $G_q$ of $q$ elements, since by F2, a prime-order group has no small subgroups other than easily-detected $G_1 = \{1\}$. A historical choice was to use $p$ of 1024 bits and a 160-bit $q$; a more conservative choice now is $p$ of 2048 bits and a 256-bit $q$.

4. A *secure prime* is a prime $p = 2Rq + 1$ such that $q$ is prime, and either $R$ is also prime or every prime factor $q_i$ of $R$ is larger than $q$, i.e., $p = 2qq_1q_2 \cdots q_n + 1$ for primes $q_i > q$. Secure primes can be generated much faster than safe primes, and are believed to be no weaker against known attacks.

Aside: the term *strong prime* is unavailable for DH duty, allocated instead for service in describing the properties of primes necessary for security in RSA moduli of form $n = pq$.

**SUBGROUP CONFINEMENT ATTACK ON DH.** Let the DH prime be $p = Rq + 1$ (assume $q$ is a large prime as required, i.e., a *PH-safe prime*). $R$ itself may be very large, but typically it will have many smaller divisors $d$ (e.g., since $R$ must be even, 2 is always a divisor). Let $g$ be a generator for $Z_p^*$. For any such $d$, $b = g^{(p-1)/d}$ has order $d$ (from F6). The attack idea is to push computations into the small, searchable subgroup of order $d$. To do this so that both parties still compute a common $K$, intercept (e.g., via a middle-person attack) the legitimate exponentials $g^a, g^b$ and raise each to the power $(p-1)/d$; the resulting shared key is $K = g^{ab(p-1)/d} = (g^{(p-1)/d})^{ab} = b^{ab}$. Since $b$ has order $d$, this key can take on only $d$ values. Such attacks highlight the importance of integrity-protection in protocols (including any system parameters exchanged).

**Exercise** (Toy example of small-subgroup confinement). Work through a tiny example of the above attack, using $p = 11$ (building on examples from above). Use $R = 2 = d$, $q = 5$, $g = 2$. The DH private keys should be in the range $[1, 9]$.

**Exercise** (Secure primes and small-subgroup confinement). Suppose that $p$ is a *secure prime*, and a check is made to ensure that the resulting key is neither 1 nor $p-1 \pmod p$. Does the above subgroup confinement attack succeed?

**Exercise** (DSA subgroups and subgroup confinement attack). Consider $p = Rq + 1$ (prime $q$), and DH using as base a generator $g_q$ of the cyclic subgroup $G_q$.
(a) If an attacker substitutes exponentials with values (integers mod $p$) that are outside of those generated by $g_q$, what protocol checks could detect this?
(b) If an attacker changes one of the public exponentials but not both, the parties likely obtain different keys $K$; discuss possible outcomes in this case.

**ESSENTIAL PARAMETER CHECKS FOR DIFFIE-HELLMAN PROTOCOLS.** The subgroup confinement attack can be addressed by various defenses. Let $K$ denote the resulting DH key, and $x = g^a$, $y = g^b \pmod p$ the exponentials received from $A$, $B$.

1. Case: DH used with a *safe prime* $p = 2q + 1$ (here a generator $g$ for $G_p$ is used). Check $x \neq 1$ or $p-1 \pmod p$; and that $x \neq 0$ (or equivalently, check $K \neq 0$). Similarly for $y$. These checks detect if $x$ or $y$ were manipulated into the tiny subgroup $G_2$ of two elements. Note: $-1 = p-1 \pmod p$.
2. Case: DH used with a *DSA prime* (and a generator $g = g_q$ for $G_q$). The above checks are needed, plus checks that $x, y$ are in $G_q$ by confirming: $x^q \equiv 1 \pmod p$, and analo-

gously for $y$. (Thus the value $q$ must also be communicated to end-parties.)

The powering check isn't needed if the protocol itself verifies integrity of exchanged values (as in STS). The extra cost for $G_q$ exponentiation, beyond an equality test for 0, 1, $-1$, is mitigated by exponent math being mod $q$ (in a DSA subgroup, $q \ll p$).

3. Case: DH used with a *secure prime* (here a generator $g = g_q$ for $G_q$ is used). Here the checks for the above DSA prime case suffice (this is a subcase), but only the checks for the safe prime case are essential—because in an untampered run using base $g_q$, exponentials remain in $G_q$; and altering exponentials cannot result in a small-subgroup attack (other than $G_2$, ruled out by the checks of the safe prime case) because every other subgroup $G_{q_i}$ is *larger* than $G_q$, because $q_i > q$.
4. Case: DH used with a protocol that detects tampering of messages (e.g., STS). This detection will then detect small-subgroup attacks, which tamper with exponentials. In all cases, integrity of the DH generator $g$ and modulus $p$ must also be ensured. As a bonus, techniques that provide end-to-end authentication also preclude successful middle-person attacks.

**SHORT DH EXPONENTS FOR EFFICIENCY.** When DSA primes are used for DH, the base used is a generator $g_q$ for $G_q$. Since exponent arithmetic is modulo the order of the base, DH exponents $a$, $b$ can thus be reduced mod $q$, making exponentiation considerably faster. In essence, *short exponents* are used—with $q$ known, private exponents $a$, $b$ are simply chosen in $[1, q-1]$, vastly smaller than $[1, p-1]$. With safe primes, one can also use private exponents of this size—but in both cases, no shorter since specialized discrete log algorithms (parallelizable with linear speedup) can find $2t$-bit exponents in $2^t$ operations. Thus for attackers capable of $2^{80}$ operations, exponents must be at least 160 bits. DSA subgroups of order $2^{256}$ are matched with $t = 128$-bit exponents.

**SAFE PRIME ADVANTAGES (SUMMARY).** The advantage of using a safe prime ($p = 2q+1$) is that the guaranteed absence of small prime factors of $p-1$ (other than 2) precludes subgroup confinement attacks, and there is no need to communicate to DH end-parties an extra parameter such as the value $q$ in the DSA prime case ($p = Rq+1$). The secure prime case conveys these same benefits. These cases can also utilize "short" DH exponents (above), e.g., choosing DH exponents $a, b$ in the range $2^{m-1} < a, b < 2^m$ if $m$-bit exponents are deemed sufficient ($m$ often equals the bitlength of a DSA prime $q$).

**KEY-USE CONFIRMATION IS NOT ENOUGH.** Confirmation by end-parties that they have agreed upon the same key, i.e., key-use confirmation, does not guarantee the absence of an attack. An attacker changing both exponentials to 0 (or 1), forces both parties to compute $K = 0$ (or $K = 1$). Thus confirming an agreed key is insufficient, and different than verifying non-tampering of the values that you believe were exchanged to get it.

**COMPARING MIDDLE-PERSON AND SMALL-SUBGROUP ATTACK.** Subgroup confinement requires an active attack during key agreement; after successful determination of the shared key, passive traffic monitoring allows decryption with the known key. In contrast, a DH middle-person attack requires active involvement during key agreement, plus ongoing inline participation (decrypting, encrypting, and forwarding data). In this respect, the subgroup attack is simpler—but also stopped by simple checks at the endpoints.

**Exercise** (Diffie-Hellman small subgroups and timing attacks discussed in RFC 7919). (a) Discuss how RFC 7919 [18] proposes to ameliorate small-subgroup attacks on TLS. (b) Discuss the attacks motivating this text in RFC 7919, Section 8.6 (Timing Attacks): *"Any implementation of finite field Diffie-Hellman key exchange should use constant-time modular-exponentiation implementations. This is particularly true for those implementations that ever reuse DHE [Diffie-Hellman Ephemeral] secret keys (so-called "semi-static" ephemeral keying) or share DHE secret keys across multiple machines (e.g., in a load-balancer situation)."*

## 4.9 ‡End notes and further reading

The highly accessible and definitive encyclopedia on authentication and authenticated key establishment is Boyd [9]. It includes discussion of protocol goals, *good keys*, and formal verification of authentication protocols including the BAN logic [12] and others. See Menezes [33, Ch.12] for a shorter systematic treatment, and also for *random numbers* and *pseudo-random number generators* (PRNGs), a topic discussed less formally by Ferguson [15]. The grandmaster postal chess attack is attributed to John Conway circa 1976. On *interleaving attacks*, Bird [8] systematically examines symmetric-key protocols, while Diffie [14] gives examples including public-key protocols. HTTP *digest authentication* is specified in RFC 7616 [41]. Bellovin [6] discusses *partition attacks* on RSA-EKE and DH-EKE. For guessing attacks on Kerberos passwords, see Wu [50]. Kerberos and many other protocols were inspired by the Needham-Schroeder shared-key protocol [34]. Also of general interest on engineering crypto protocols and what goes wrong, see Abadi [1] and Anderson [3]. For a survey on secure *device pairing methods*, see Kumar [27].

Both EKE [6] and SPEKE [24] require that verifiers store cleartext passwords; password file compromise is mitigated by augmented versions [7, 25]. Steiner [42] proposed the minimal three-message version of EKE. High-profile alternatives include SRP [49, 48] and J-PAKE [22, 20]. For an independent analysis of J-PAKE, see Abdalla [2]. Patel [38] discusses subtle active attacks on EKE. For security concerns about SPEKE, see Hao [23]. Boyd [9, Ch.7] authoritatively summarizes the abundant academic analysis of PAKE protocols; Kaufman [26] lucidly overviews a subset. The three examples in Section 4.6 follow Gong [19], who also pursues a formal definition of *verifiable text*. The definition of *forward secrecy* is from Diffie [14], which elucidates the STS protocol, and the general notion that legitimate end-parties have matching protocol transcripts in secure protocol runs. For *unknown key-share attacks* see Boyd [9] and Law [28]. Many variants of DH with long-term secret keys exist ([33, p.515–519], [28]), but non-ephemeral DH variants have a poor track record [29] against small-subgroup attacks.

Neuman [35] overviews Kerberos V5; see also RFC 4120 [36]. Section 4.7 is based on Menezes [33, p.501]. Pashalidis [37] gives an SSO taxonomy. Wang [47] explores flaws in web SSO including OpenID; see Mainka [32] for OpenID Connect, NIST SP 800-63C [39] for federated identity systems, and Chiasson [13] for password managers. Section 4.8 background is based on Menezes [33]; page 19 thereof defines *strong prime*.

Schnorr's signature scheme [40] used prime-order subgroups prior to the later-named *DSA primes* in NIST's Digital Signature Algorithm. Regarding discrete log algorithms, see van Oorschot [46, 45] respectively for general parallelization with linear speedup, and to find exponents of size $2^{2t}$ (i.e., $2t$ bits) in order $2^t$ operations. Regarding "trap-dooring" of a 1024-bit prime $p$ and taking a Diffie-Hellman log in such a system, see Fried [16]. Small-subgroup attacks were already published in 1996 [33, p.516]. Simple checks to prevent them, and corresponding checks before issuing certificates, were a prominent topic circa 1995-1997 (cf. [51]). Early papers highlighting that Diffie-Hellman type protocols must verify the integrity of values used in computations include: Anderson (with Needham) [4], van Oorschot [45], Jablon [24], Anderson (with Vaudenay) [5], and Lim [29]. A 2017 study [43] found that prior to its disclosures, such checks remained largely unimplemented on Internet servers. For algorithms to efficiently generate primes suitable for Diffie-Hellman, RSA and other public-key algorithms, see Menezes [33, Ch.4], and also Lim [29] for defining and generating *secure primes*.

# References


[1] M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996. See also (same authors and title): *IEEE Symp. Security and Privacy*, page 122–136, 1994.

[2] M. Abdalla, F. Benhamouda, and P. MacKenzie. Security of the J-PAKE password-authenticated key exchange protocol. In *IEEE Symp. Security and Privacy*, pages 571–587, 2015.

[3] R. J. Anderson and R. M. Needham. Programming Satan's Computer. In *Computer Science Today: Recent Trends and Developments*, pages 426–440. 1995. Springer LNCS 1000.

[4] R. J. Anderson and R. M. Needham. Robustness principles for public key protocols. In *CRYPTO*, pages 236–247, 1995.

[5] R. J. Anderson and S. Vaudenay. Minding your p's and q's. In *ASIACRYPT*, pages 26–35, 1996.

[6] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Symp. Security and Privacy*, pages 72–84, 1992.

[7] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM Comp. & Comm. Security (CCS)*, pages 244–250, 1993.

[8] R. Bird, I. S. Gopal, A. Herzberg, P. A. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of two-party authentication protocols. In *CRYPTO*, pages 44–61, 1991.

[9] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003. Also second edition (2019) with Douglas Stebila.

[10] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus. NIST Special Pub 800-63-1: Electronic Authentication Guideline. U.S. Dept. of Commerce. Dec 2011 (121 pages), supersedes [11]; superseded by SP 800-63-2, Aug 2013 (123 pages), itself superseded by [39].

[11] W. E. Burr, D. F. Dodson, and W. T. Polk. NIST Special Pub 800-63: Electronic Authentication Guideline. U.S. Dept. of Commerce. Ver. 1.0, Jun 2004 (53 pages), including Appendix A: Estimating Password Entropy and Strength (8 pages). Superseded by [10].

[12] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990. See also (same authors and title) *ACM SOSP*, pages 1–13, 1989.

[13] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security*, 2006.

[14] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[15] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.

[16] J. Fried, P. Gaudry, N. Heninger, and E. Thomé. A kilobit hidden SNFS discrete logarithm computation. In *EUROCRYPT*, pages 202–231, 2017.

[17] K. Gaarder and E. Snekkenes. Applying a formal analysis technique to the CCITT X.509 strong two-way authentication protocol. *Journal of Cryptology*, 3(2):81–98, 1991.

[18] D. Gillmor. RFC 7919: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS), Aug. 2016. Proposed Standard.

[19] L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE J. Selected Areas in Commns*, 11(5):648–656, 1993.

[20] F. Hao. RFC 8236: J-PAKE—Password-Authenticated Key Exchange by Juggling, Sept. 2017. Informational.

[21] F. Hao and P. Ryan. Password authenticated key exchange by juggling. In *2008 Security Protocols Workshop*, pages 159–171. Springer LNCS 6615 (2011).

[22] F. Hao and P. Ryan. J-PAKE: Authenticated key exchange without PKI. *Trans. Computational Science*, 11:192–206, 2010. Springer LNCS 6480.

[23] F. Hao and S. F. Shahandashti. The SPEKE protocol revisited. In *Security Standardisation Research (SSR)*, pages 26–38, 2014. Springer LNCS 8893. See also *IEEE TIFS* (2018), "Analyzing and patching SPEKE in ISO/IEC".

[24] D. P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996.

[25] D. P. Jablon. Extended password key exchange protocols immune to dictionary attacks. In *Workshop on Enabling Technologies/Infrastructure for Collaborative Enterprises (WET-ICE)*, pages 248–255, 1997.

[26] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communications in a Public World (2nd edition)*. Prentice Hall, 2003.

[27] A. Kumar, N. Saxena, G. Tsudik, and E. Uzun. Caveat emptor: A comparative study of secure device pairing methods. In *IEEE Pervasive Computing and Comm. (PerCom 2009)*, pages 1–10, 2009.

[28] L. Law, A. Menezes, M. Qu, J. A. Solinas, and S. A. Vanstone. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography*, 28(2):119–134, 2003.

[29] C. H. Lim and P. J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *CRYPTO*, pages 249–263, 1997.

[30] S. Lucks. Open Key Exchange: How to defeat dictionary attacks without encrypting public keys. In *Security Protocols Workshop*, pages 79–90, 1997.

[31] P. D. MacKenzie, S. Patel, and R. Swaminathan. Password-authenticated key exchange based on RSA. In *ASIACRYPT*, pages 599–613, 2000.

[32] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich. SoK: Single sign-on security—An evaluation of OpenID Connect. In *IEEE Eur. Symp. Security & Privacy*, pages 251–266, 2017.

[33] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Free at: `http://cacr.uwaterloo.ca/hac/`.

[34] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Comm. ACM*, 21(12):993–999, 1978.

[35] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–38, Sept. 1994.

[36] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. RFC 4120: The Kerberos Network Authentication Service (V5), July 2005. Proposed Standard; obsoletes RFC 1510.

[37] A. Pashalidis and C. J. Mitchell. A taxonomy of single sign-on systems. In *Australasian Conf. on Info. Security & Privacy (ACISP)*, pages 249–264, 2003.

[38] S. Patel. Number theoretic attacks on secure password schemes. In *IEEE Symp. Security and Privacy*, pages 236–247, 1997.

[39] Paul A. Grassi et al. NIST Special Pub 800-63-3: Digital Identity Guidelines. U.S. Dept. of Commerce. Jun 2017, supersedes [10]. Additional parts SP 800-63A: Enrollment and Identity Proofing, SP 800-63B: Authentication and Lifecycle Management, SP 800-63C: Federation and Assertions.

[40] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[41] R. Shekh-Yusef, D. Ahrens, and S. Bremer. RFC 7616: HTTP Digest Access Authentication, Sept. 2015. Proposed Standard. Obsoletes RFC 2617.

[42] M. Steiner, G. Tsudik, and M. Waidner. Refinement and extension of encrypted key exchange. *ACM Operating Sys. Review*, 29(3):22–30, 1995.

[43] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Netw. Dist. Sys. Security (NDSS)*, 2017.

[44] P. C. van Oorschot. Extending cryptographic logics of belief to key agreement protocols. In *ACM Comp. & Comm. Security (CCS)*, pages 232–243, 1993.

[45] P. C. van Oorschot and M. J. Wiener. On Diffie-Hellman key agreement with short exponents. In *EUROCRYPT*, pages 332–343, 1996.

[46] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[47] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symp. Security and Privacy*, pages 365–379, 2012.

[48] T. Wu. RFC 2945: The SRP Authentication and Key Exchange System, Sept. 2000. RFC 2944 (Telnet) and RFC 5054 (TLS) rely on SRP; see also `http://srp.stanford.edu/` (Stanford SRP Homepage).

[49] T. D. Wu. The secure remote password protocol. In *Netw. Dist. Sys. Security (NDSS)*, 1998.

[50] T. D. Wu. A real-world analysis of Kerberos password security. In *Netw. Dist. Sys. Security (NDSS)*, 1999.

[51] R. Zuccherato. RFC 2785: Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME, Mar. 2000. Informational.