Applied Cryptography and Network Security

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Strong Password Protocols

University of Pittsburgh

Today's Outline

Now, we'll focus on strong password protocols

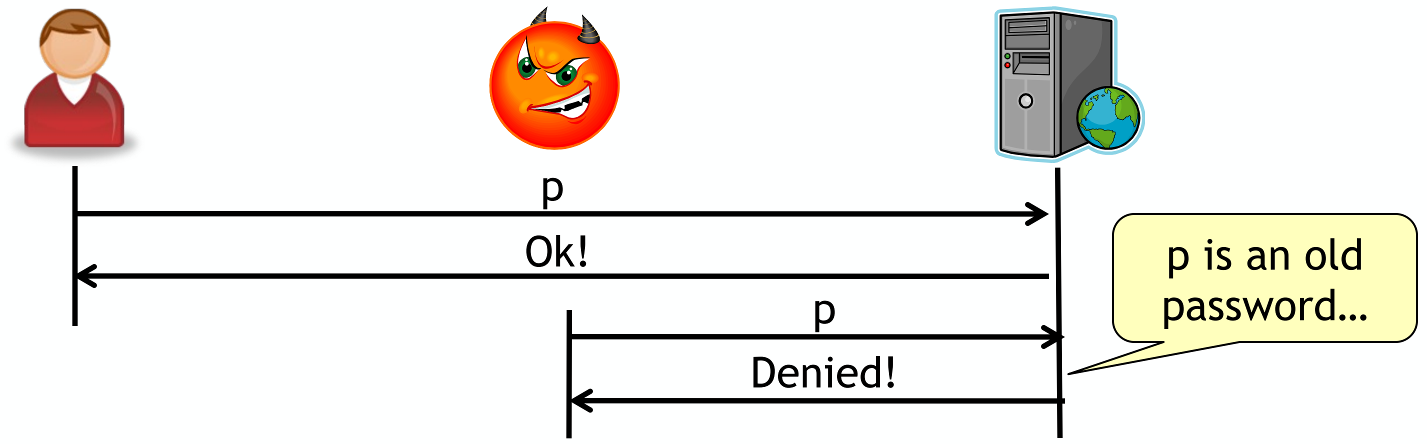
In particular, we'll look at

- Lamport's hash-based one-time password scheme
- Encrypted Key Exchange (EKE)
- Secure Remote Password (SRP)
- Secure credential download protocols

As we'll see, these protocols allow us to leverage weak passwords into strong cryptographic protocols

One problem with password-based systems is that if the password is ever observed, it is compromised

In a one-time password scheme, passwords are invalidated after use



Clearly, this prevents impersonation attempts by a passive adversary

However, these systems come at a cost

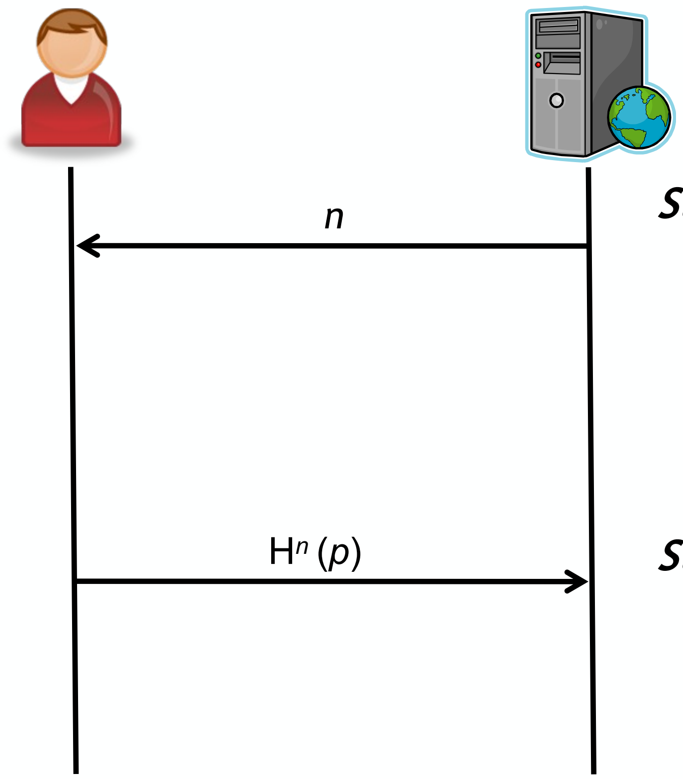
- Do you *really* expect users to memorize a list of passwords?
- Will this require that the server stores tons of state for each user?
- …

It turns out that these types of systems are actually quite easy to deploy!

Leslie Lamport developed a one-time password scheme that uses hash chains

Leslie Lamport, "Password Authentication with Insecure Communication," Communications of the ACM 24(11):770-772 November 1981.

Setup Phase



Step 1:

- Choose a large n (e.g., $n = 1000$)
- Send n to the user

Step 2:

- Choose a password p
- Compute $H^n(p)$
- Send $H^n(p)$ to server
- Store (p, n)

Step 3:

- Store $(n, H^n(p))$

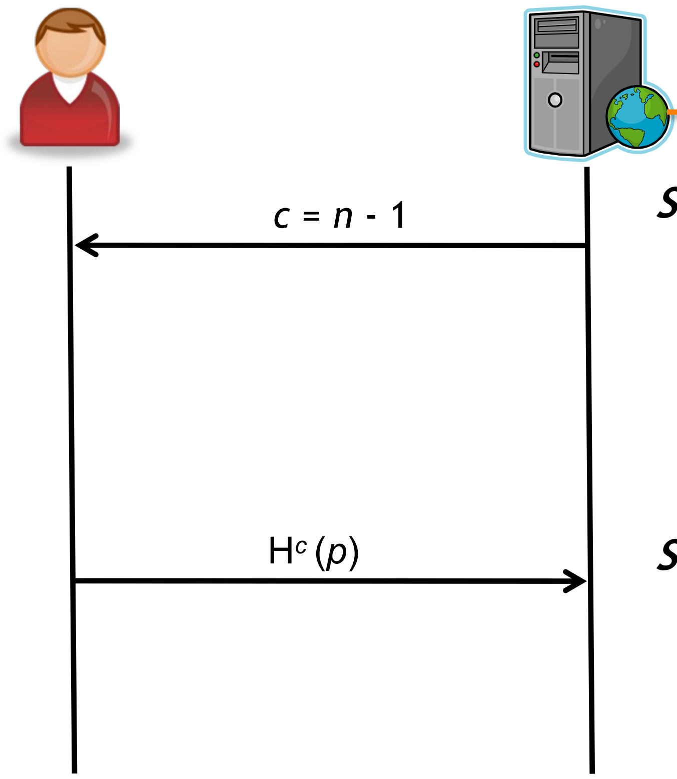
Notation

- H is a hash function
- $H^n(p)$ represents n applications of H to p
- E.g., $H^2(p) = H(H(p))$

Using Lamport's OTP Scheme

~~Storage: (p, n)~~

~~Storage: (n, $H^n(p)$)~~



Step 1:

- Send c = n - 1 to user

Step 2:

- Verify that $c < n$
- Compute $H^c(p)$
- Send x = $H^c(p)$ to the server
- Store (p, c)

Step 3:

- Verify that H(x) = $H^n(p)$
- Store (n - 1, $H^{n-1}(p)$)

Storage: (p, c)

Storage: (n-1, $H^{n-1}(p)$)

Why is this scheme safe?

To prove the safety of these scheme, we need to show that knowing an old (challenge, response) pair does not help the attacker

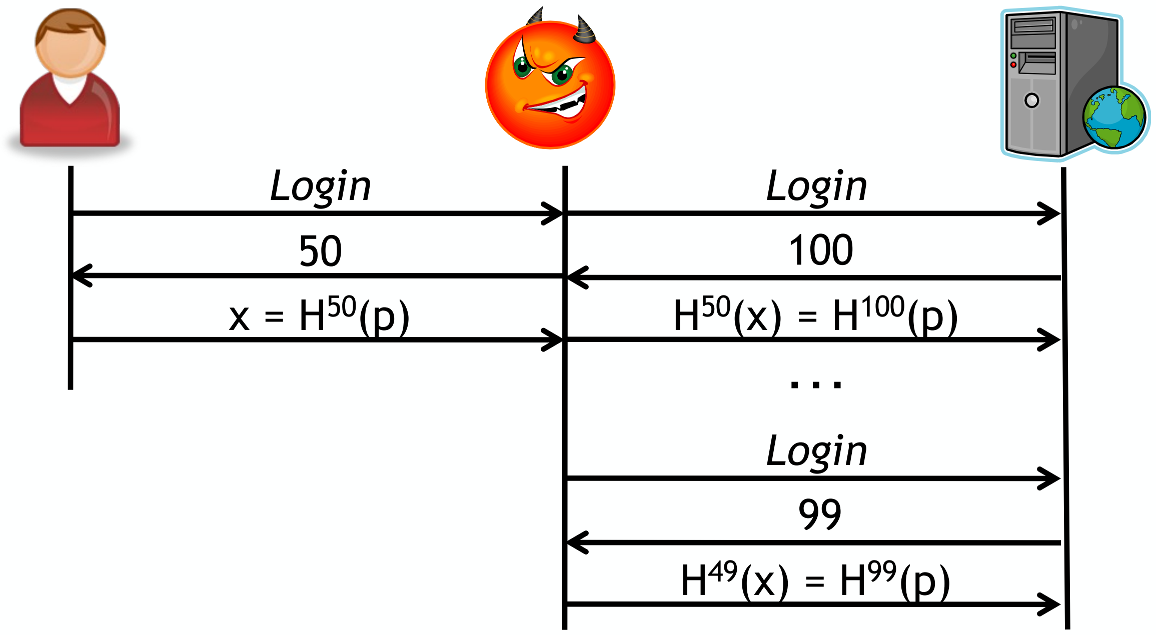
Attack 1: The adversary attempts to use an old (challenge, response)

- The user will never accept an old challenge
- The server will never request an old challenge (*n* decremented after use)

Attack 2: Derive the k^{th} password from the $k-1^{st}$ password

- Assume the $k-1^{st}$ password is $H^m(p)$
- This means that the k^{th} password will be $H^{m-1}(p)$
- To guess the k^{th} password, we need a value v such that $H(v) = H^m(p)$
    - That is, we need to find the preimage of $H^m(p)$
- The *preimage resistance* property of H means that this is infeasible

Lamport's scheme is not secure against an active attacker (meddler in the middle)



The adversary does not know p, but can impersonate Bob anyway!

Question: Can we simply require that challenges decrement by 1?

- What about packet loss?
- Failed login attempts by others?

In short, this system will only work if our deployment environment assumes that there are no active attackers

Strong Password Protocols

Strong password protocols are designed to prevent both passive and active attackers from gaining enough information to conduct an offline password cracking attempt

This class of protocols was first proposed by Bellovin and Merritt

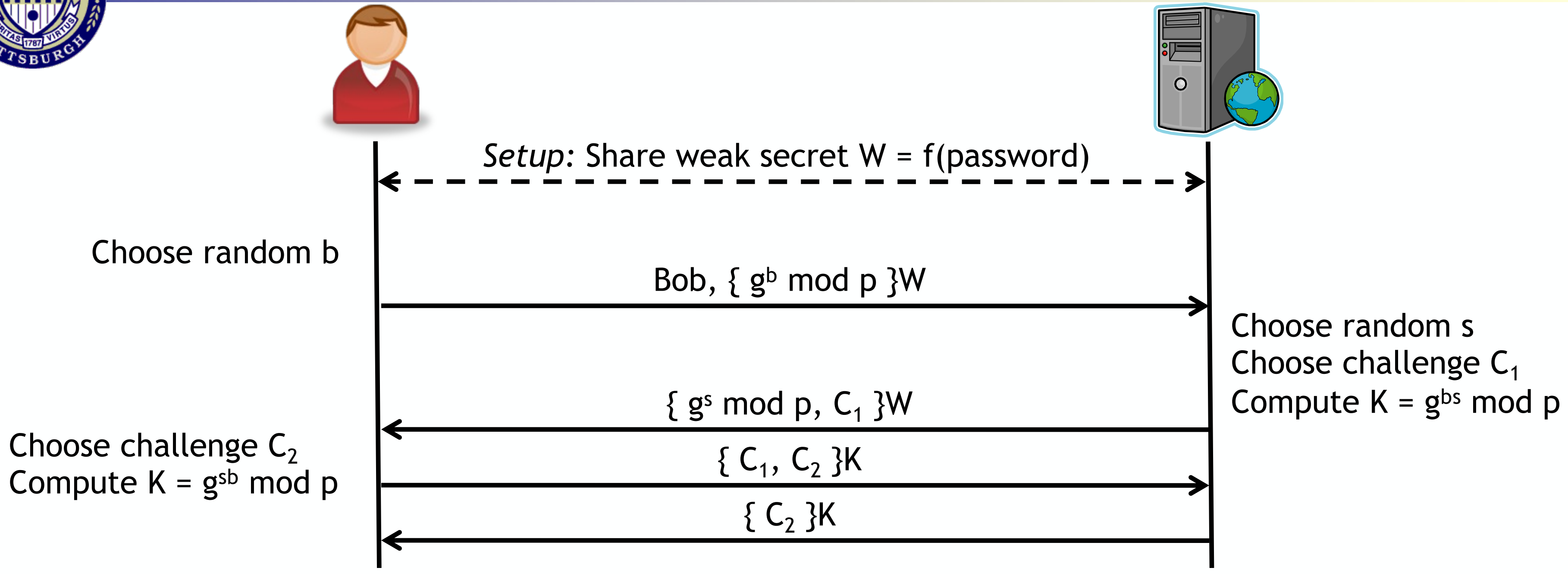
- Encrypted key exchange (EKE)

At a high level this protocol works as follows:

- Bob and the server share some weak secret (i.e., a password)
- Both parties carry out a Diffie-Hellman key exchange
  - Messages encrypted using the weak secret
- Mutual authentication occurs using the D-H key

This works because the whole exchange essentially looks random to outside observers!

EKE in Detail



How does this protocol prevent offline password guessing?

- Decrypting { g^b mod p }W using the wrong secret gives a randomized output
- Further, g^b mod p is essentially a random number mod p
- As a result, the result of properly decrypting { g^b mod p }W also looks randomized if b is unknown
- Result: There's no way to "check" whether W' = W for a password guess W'

Interestingly, our choice of modulus p can actually make it possible for adversaries to attack this protocol!

Observation: g^b mod p < p by the definition of "mod"

If an adversary decrypts { g^b mod p }W using a guess W' and obtains a value greater than p, then W' is certainly not the correct secret

This could be a problem if p is slightly greater than a power of 2

Why? Assume p slightly bigger than 2^n

- The binary representation of p requires n+1 bits
- Since $2^{n+1} = 2 \times 2^n$, this bit field can hold (roughly) two times as many values as are actually needed by the protocol (valid mod p)
- As such, a random decryption has (roughly) a 1 in 2 chance of being greater than the value p

What's the fix? Choose a p that is slightly less than a power of 2

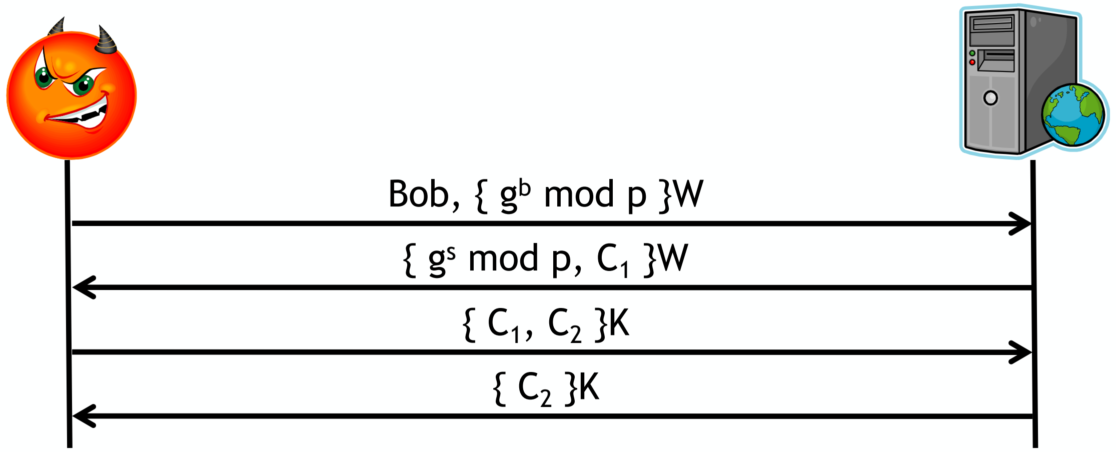
- In practice, try to "fill" the last block as fully as possible

What happens if the server is compromised?

For EKE to work, the server needs to store a list of 〈user, W〉 bindings

If the server is compromised, the adversary can impersonate any user!

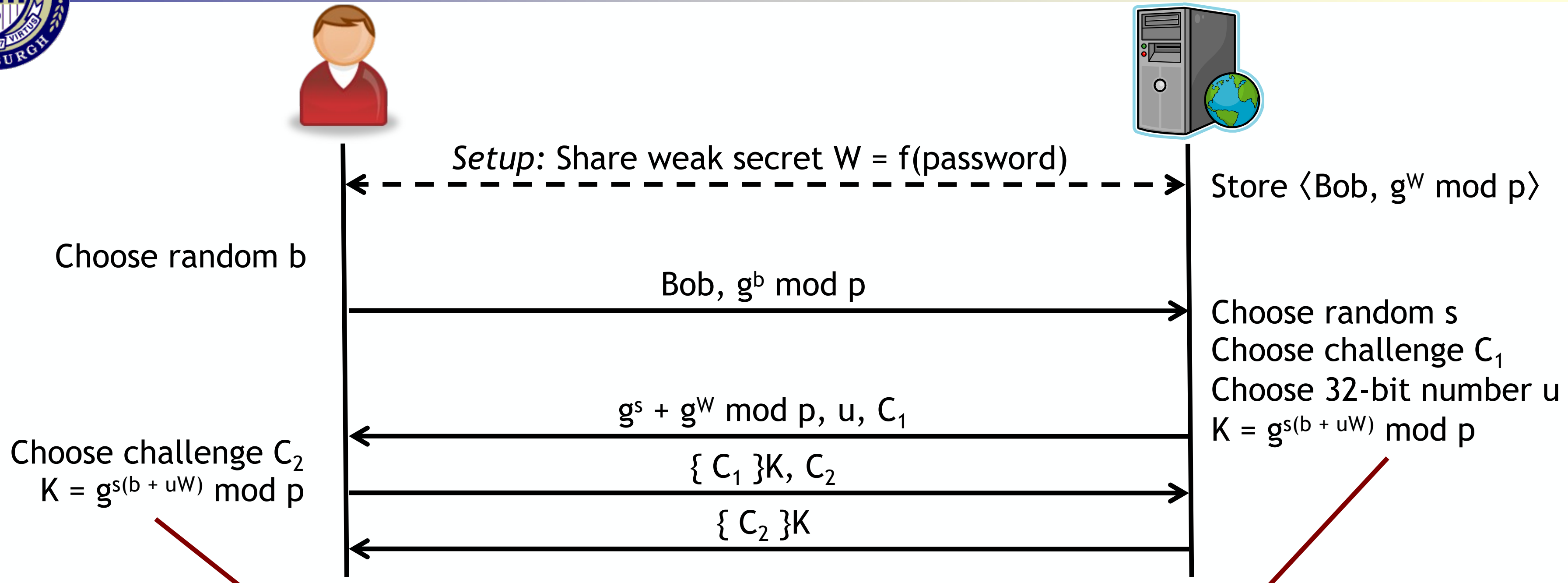
- The adversary doesn't know the password, but they don't need it
- W is all that is needed to authenticate!



Ideally, this shouldn't happen…

New property: Compromising the server should still require the adversary to launch a dictionary attack to recover W

The Secure Remote Password (SRP) protocol provides us with this assurance



Given W, b, u, $g^s + g^W$ mod p
- $g^s + g^W$ mod p – g^W mod p = g^s mod p
- $(g^s)^{(b + uW)}$ mod p = $g^{s(b + uW)}$ mod p

Given g^b mod p, g^W mod p
- $(g^W)^u$ mod p = g^{uW} mod p
- g^b mod p × g^{uW} mod p = g^{b+uW} mod p
- $(g^{b+uW})^s$ mod p = $g^{s(b+uW)}$ mod p

Question: Why does SRP force the adversary to launch a dictionary attack?

Aren't passwords old technology? Why are we learning about this?

Asymmetric key cryptography seems like a much cooler solution... Can't we just use this for authentication?

For this to work, we need to manage public and private keys

- Public keys can be stored publicly, so this is no problem
- Where do we keep our private keys?

What if I need to use multiple machines? Replicating secrets is bad. Plus, I don't trust my administrators.

Private keys can be stored in a number of ways

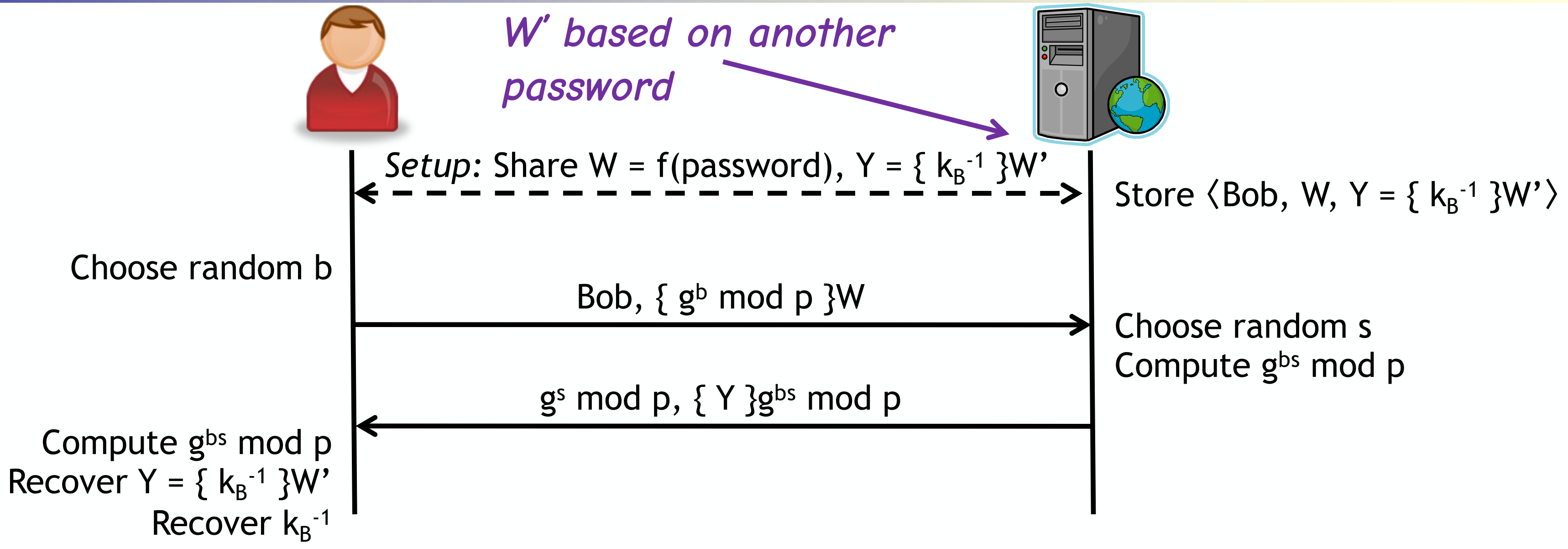
- On the local machine, protected by the file system's permission settings
- On a smart card or some other token
- On a trusted server

What if I lose or break my token?

What if the server is compromised? What if I don't trust my trusted server?

Strong password protocols can help us solve the private key storage conundrum in a robust fashion

We can safely store our private keys so that even if the server is compromised, the key is not leaked!


W' based on another password
Setup: Share W = f(password), Y = { k_B^{-1} }W'
Store 〈Bob, W, Y = { k_B^{-1} }W'〉
Choose random b
Bob, { g^b mod p }W
Choose random s
Compute g^{bs} mod p
g^s mod p, { Y }g^{bs} mod p
Compute g^{bs} mod p
Recover Y = { k_B^{-1} }W'
Recover k_B^{-1}

Note that the server never finds out whether Bob knew the password W

- Why? Bob never talks to the server after retrieving the encrypted key!

An attacker impersonating Bob cannot launch an offline attack against

- Message 1 commits the attacker to a single password guess
- If this guess is incorrect, the rest of the math fails to work!

Question: Why does Bob need to use two different passwords?

Conclusions

Although passwords are ancient technology, they are still widely used

Today we have discussed

- One-time password schemes that are resilient to eavesdropping attacks
- Strong password protocols that prevent offline password guessing attacks
- Hardened versions of these protocols that are also resilient to server compromise
- Secure credential retrieval protocols that allow us to use passwords to protect stronger cryptographic secrets like private keys

In the end, we'll probably never fully get rid of passwords ☹

At least these protocols allow use to use passwords in a safer manner ☺

Next time: Trusted intermediaries and Kerberos