PROFESSOR: All right, guys. So let's get started. Welcome back from what I hope was an exciting holiday for everyone. So today we're going to talk about user authentication. So the basic challenge that we want to address today is how can human users prove their identity to a program? In particular, the paper that was assigned for today's class addresses an existential question in the security community. Is there anything better than passwords for authentication?

So at a high level it seems like passwords are a terrible idea. So they have very low entropy, its very easy for attackers to guess them. Also the security questions that we use to recover from lost passwords often have even lower entropy than the passwords themselves, which also seems like a problem.

And even worse, users typically will use the same password across a lot of different sites. So that means that the vulnerability in one password, if it's easy to guess, could expose a user's activity across a wide range of sites.

So as the paper for today's class states, I love this quote, "the continued domination of passwords over all of the methods of in-user authentication is a major embarrassment for security researchers." All right, so the community just seething out there, they want some better alternative. But it's not clear if there actually is an authentication scheme that actually totally dominates passwords, that's more usable, that's more deployable, that's more secure.

So in today's lecture, we'll basically do three things. So first of all, we're going to look and we're going to see how current passwords can work. Then we're going to talk about the desirable properties at a high level for any authentication scheme.

And then we're finally going to look at what the paper gives us in terms of metrics for authenticating authentication schemes, and we're going to see how some of these other authentication schemes actually compared to passwords.

So in [INAUDIBLE] what is a password? So a password is a secret that is shared between a user and a server. So the naive implementation of a password scheme is to basically just have a table on the server side that essentially just maps user names to passwords.

That's the simplest way for you to imagine implementing one of the authentication schemes-- user passes into their user name and the password, server network does a look up in this table, compares the password of the client supplied, what's in here. If everything's good, the user's authenticated.

So clearly the problem with this is that if the attacker compromises the server, then he can just look at this table and then get all the uses passwords in the queue. So that's clearly a bad thing. So perhaps an improved solution is to have the server store a table that looks like. So once again, it'd match the user name but now it actually match to hash of the password.

So user client's gonna supply their clear text password to the server, the server will then take that clear text password, hash it, do look at the table, and once again see if the user is who he or she says that they are.

So the advantage of this scheme is that by designed these hash functions are difficult to invert. So if this table is lost, it's leaked somehow or the attacker compromised the server, and the attacker could look at these things here, but it's difficult for the attackers to say, OK, this sort of string of random alpha numeric characters here. Here's a pre-image that was used as the input of the hast function [INAUDIBLE] that value there.

So that at least is the nice thing about these hashes in theory. Now in practice, attackers don't actually have to launch brute force attacks to figure out what the preimages for these hash values are.

So attackers can actually take advantage of the fact that passwords in practice have skewed distribution. And by skewed distributions, I mean that-- let's say that we knew that all passwords were 20 characters long.

It's not like users actually pick passwords that's sort of exist in all places in that space of 20 possible characters. In practice, people pick passwords like 1, 2, 3 or todd or things like this. So in fact there's been these empirical studies of how passwords work and a lot of times these studies find things like the top 5,000 passwords cover about 20% of users.

So what that means, in other words, is that the attacker has a database of those 5,000 passwords. The attacker can just hash those, and then when the attacker looks at this stolen password table, can just see if one of those things that come from this 5,000 large list match over here. And so empirically speaking, the attacker would be able to recover about 20% of passwords that way.

And so, folks at Yahoo found that passwords have roughly 10 to 20 bits of intricate, 10 to 20 bits of randomness in them. And that's actually not that big. So, for example, if you think about what might this hash function here be? So maybe it's something like shop, something like this. So modern machines actually calculate millions of these hashes every second.

So the fact that hash function by design are suppose to be easy to calculate so it'd be fast calculate. Combined with this fact that there'd be skewed password distributions, means that in principle, this scheme here is not as secure as it might seem.

So one thing you can imagine to try to make life more difficult on the attacker is you could imagine that you use expensive key derivation function. And so by key derivation function, I just mean this thing up here. This thing that's taking the passwords as input and then generate something that's stored on the server.

So what's nice about these key derivation functions is it actually have tunable cost. So you can basically turn this knob and make that function run slower or faster depending on what you want. And so the idea here is that, let's say that you're going to use a key derivation function.

So assume these examples are like PBKDF2, or maybe BCrypt so you can look these up using the miracle of the internet if you care to know more about them. But the base idea is let's imagine that one of these key derivation function took a second to calculate, as opposed to a few milliseconds.

That actually makes the attacker's job much more difficult. Because when the attacker is trying to, let's say, generate values for these 5,000 topmost passwords, it's going to take the attacker much longer to do that.

So does that all makes sense how these things work? Pretty straight forward. So internally these key derivation functions often operate by repeatedly calling a hash multiple, multiple times. So that's all pretty straightforward.

So you might say, well, does this solve the problem? So can we just use these expensive key derivation function and be done with it? So if this was a security class, the answer is no. So one problem is that the adversary can build something called rainbow tables. And so a rainbow table is basically just a map of a password to hash out.

And so the insight here is that even if the system is using one of these expensive key derivation function, the attacker can calculate one of these tables once. It might be a little bit painful because each key derivation function indication is slow. But the attacker can build this table once and then use that to crack all subsequent systems the attacker can break into that use that same key derivation function. So that's how rainbow tables work.

And once again, to maximize the cost benefit of building this rainbow table, the attacker could take advantage of the skewed password distributions I can see up here. So the attacker might only build a rainbow table for some small set of all possible passwords.

AUDIENCE: So salting makes this much more difficult.

PROFESSOR: Yeah, yeah, that's right. So we're going to get to salting I believe in a couple seconds. That's right. So at a high level, if you don't use salting, rainbow tables actually allow the attacker to spend some effort offline, calculate this table, and then sort of amortized the cost of calculating that table over breaking many different password databases.

So the next thing that we can think about to improve things is salting. I swear that guy was not a plant, I will give you your $20 after class. So how does salting work? So the basic thing is you just want input some additional randomness into the way that the passwords generated.

So basically, you want to take this hash function and you want to put some salt in there-- which I'll explain in a second-- and then the password. And this is the thing that you saw on the server side in the [INAUDIBLE].

So what is this salt? And you just think of it as just a string, a long string that's provided as sort of a first part to this hash function. So why is it better to use this scheme? And know that the salt is actually stored on the clear text on the server side.

So you might be thinking OK, well if that salt is stored on the clear text in the server side, it seemed like a server can both steal the table that matched user names to passwords and the attacker can also steal the salt. So why is that useful?

AUDIENCE: Because if you picked the top most common password, you can't just use it once and find a new user.

PROFESSOR: That's exactly right. So basically what this does is this prevents the attacker from building a single rainbow table and then using that rainbow table against all instances of that hash function. And so you can basically think of this as sort of uniquifying passwords even if they are the same, basically.

So this is what a lot of systems do in practice, they use this notion of salt here. And so the best practices for this so you want to choose a salt that's long Because you're going to essentially think of the salt as adding more bits to this pseudopassword right. So more bits is always better.

And the other thing you want to do to is that whenever the user changes his or her password, you typically want to change that salt too. So one reason for that is let's say that users are lazy and they want to pick the same password multiple times. Changing the salt will ensure that the thing that's stored in the password database will actually be different even it that password's the same. I think there was a questions somewhere.

AUDIENCE: Why's it called salt?

PROFESSOR: I'm actually not sure why it's called salt, that's a good question. I'm sure there's some answer to this though. It's like why are cookies called cookies? The internet will know but I actually don't know.

AUDIENCE: Add some [INAUDIBLE] to the hash number hash [INAUDIBLE].

PROFESSOR: There we go. I'm glad that we're getting this on film, cause I feel this how we're going to get our Touring awards. That's right. I'm sure there's some answer on the internet, so I'll look that up later. But does that all basically makes sense?

OK so these approaches are fairly straightforward. So what I've assume so far is that somehow the client is transmitting the password to the server. But I haven't actually specified how that transition's actually going to take place.

So how do we transmit these passwords? So the first idea you might have would be, well, we'll just send the password in the clear over the network. This is clearly cartoonishly bad, because then there could be a network attacker who's basically snooping and seeing the traffic that you're sending. And let's see if we can just take that password right off the wire and then impersonate you.

So we always start with the straw man before I show you the other straw men, which of course are also fatally flawed. So first thing you think about is sending a password in the clear. Another thing you might think, which would be a little bit better perhaps, is perhaps we send the password over an encrypted connection.

And so we use some type of cryptography here. Maybe there's some secret key or something like that and that's what we use to transform the password before we send it over the connection. So at a high level, encryption always seems to make things better, right? Trademark.

But the problem is that unless you think carefully about how you're using things like encryption and hashing, you may not be getting the security benefits that you think you're getting. Because, for example, what if there's someone who's sitting between you-- the client-- and the server, this proverbial man in the middle attacker, who's actually snooping on your traffic and pretending to be the server.

If you send encrypted data, you haven't actually authenticated the other end, then you could still be opening up yourself to problems. Because if the client just, let's say, picked some random key, sends it to some entity on the other side who may or may not be the server. It is not the server, [INAUDIBLE]. You are sending something to some person, who will then be able to get all your secrets.

And so similarly, people might think well what if I don't send the raw password but I send a hash of the passwords. That actually doesn't give you anything in and of itself either. Because whether you send the password or the hash of a password-- I mean, a hash of the password has the same sort of semantic power as the original password itself. If you haven't authenticated the other side if you haven't authenticated the server or things like this.

So the basic point with this discussion here is just to stress the fact that just adding encryption or just adding hashing doesn't necessarily give you any additional powers. If the client can't authenticate who he or she is sending the password to then the client could be mistakenly divulging that password with someone they don't intend to divulged it to.

So perhaps a better idea than these two is to use what they call a challenge response protocol. And here's an example of a very simple challenge response protocol. So let's say we've got the client here, and then you've got the server over here.

So the client says, hi, I'm Alice. And then the server response with some challenge seam, some quantity that the server got to pick. And then the client is going to respond with the hash of that server sent challenge, and then you can concatenate that with the password.

So at this point, the server can take this quantity. The server knows the challenge that it sent. And presumably the server knows the password, so the server can [INAUDIBLE] this quantity and see it actually matches what the user sent.

So what's nice about this protocol is that if we ignore man in the middle attacks for a second, the server is now confident that the user's actually Alice, because only Alice would know this password here. And what's nice about this is that if the server is actually the attacker-- so in other words, if Alice sent this thing to someone who's not the person who she's trying to authenticate to, then the attacker still doesn't know the password.

Because the attacker got to choose C, but the attacker doesn't know what this is. And so basically for the attacker to figure out what the password is, the attacker has to be able to, once again, invert these hash functions. Do you have a question?

AUDIENCE: I'm just curious, how can you not make a client do the hashing? [INAUDIBLE].

PROFESSOR: So let's see, so your proposed scheme is that the client side is going to call this thing?

AUDIENCE: Yeah, so instead of setting the password, and having the server hash the password and check it, the client would just send the hash password.

PROFESSOR: The client would just sent the hash password. So there's a couple reasons. So one reason, as we'll discuss later, is that there's going to be things called anti-hammering defenses right. Anti-hammering defenses is designed to prevent a bad client from continually asking, is this the password, is this the password, is this the password?

So then as a result, it's easier for things to be on the server side as on the client side. But suffice it to say, you can, in fact, do the hash on the client side. Using JavaScripts or something like this. But the basic idea is that somehow you have to have the computational expense be very, very large, because that's going to prevent the attacker from just guessing what the password is quickly. Is there another question?

AUDIENCE: Well I just wanted to point out that if the client does the hashing, then it's [INAUDIBLE] because your password is the hash.

PROFESSOR: So that's true.

AUDIENCE: So if somebody get the table from the server [INAUDIBLE] using it to hash they can log in.

PROFESSOR: That's right. Yeah, it gets a little bit subtle sometimes depending on who can pick, for example, these challenge values. Because if client and servers can pick challenge values, so that makes it more or less difficult for the client to launch those types of attacks.

So for example, like one problem with this protocol here is that basically the client doesn't get to inject any randomness into this. So you can imagine that you can make this protocol more difficult for the server to invert. If the client actually got to choose some challenge that was put in here, so you got the server side challenge verses the client side challenge. But you're right about that.

Any other questions? OK. So yeah, so this segues is discussion we're just having. So even though to break this, the server would have to invert this hash, the attacker could still try to do one of these brute force attacks. So one way that we can prevent the server from doing these brute force attacks is to choose one of these expensive hash functions like we were discussing before.

Another thing, as we just discussed, is that you could actually allow the client to, for example, choose its own client chosen challenge over here. And so that essentially would act as like a client chosen salt. So that would essentially make it more difficult for the hacker to do things like build up a rainbow table.

Because note that if the servers is the attacker here, the server always can pick the same challenge value again, again, and again, allowing to build the rainbow table. But if when the client responded back, the client also included some salt, some client chosen challenge that it included, then they'll prevent the attacker from building one of the rainbow tables. So does that all make sense? OK.

So yeah, one thing that I mentioned that might be useful to do is implementing these anti-hammer defenses. And so anti-hammering defenses are basically designed to rate limit the number of password guesses that a bad client can issue.

Because the idea here is that if you've got some clients who's trying to launch one of these brute force guesses against the password, you don't want that client to be able to sit there in a tight loop and just say, is this the password, is this the password, is this the password?

So one way we can do anti-hamming it just do that rate limiting. So the server will say, I will only accept let's say three password guesses per second from any particular client. You could also mention imagine implementing timeouts here.

So maybe the client can issue a bunch of password requests in a row, but then after, let's say, 10 of them are wrong, the server says, OK you got to hold on, I will not accept any more requests from you for, let's say, 10 seconds, something like that.

And so both of these things are designed for preventing brute force attacks. And so, for example, like some smart cars have these types of defenses, some TPNs have these kinds of defenses to basically stop against this brute force attack.

So why is it important for you to use these anti-hammering defenses? Well one reason why it's important is as we discussed these passwords have so little entropy. So because passwords typically have so little entropy, it's really important to prevent the attacker from just trying to cycle through that low entropy space very, very quickly.

So as you may be aware, a lot of websites have these format constraints that push upon you for your passwords. They'll say things like your password must have a punctuation, it must have a mixture of numbers and letters, you must have uppercase and lowercase stuff, so and so forth.

And so what those constraints are trying to get you to do is they're trying to get you to expand the entropy of the password. But what's problematic though is that it's not really these formatted constraints that we should be caring about. It's the actual entropy of the password itself.

So it turns out even if people were given these constraints-- like you have to use punctuation, characters, and stuff like that-- the entropy of resulting password is often quite low. So for example, people will often put punctuation at the beginning or end. Because they don't want to be troubled to remember like, do I have like a dollar sign in the middle or something?

And so as it turns out, these format requirements oftentimes don't make dictionary attacks much harder for a sophisticated adversary. And the reason is because, basically, the dictionary attacker can leverage these observations about how people pick passwords even in the presence of constraints.

So for example, if the attacker knows that people typically put punctuation at the beginning or the end, just incorporate that into your dictionary attack. And so an actually really interesting website you can go to that's called Telepathwords.

And so what's neat about this site is that it has a little text box. So you can type a character into that text box-- you're pretending that you're entering a password-- and Telepathwords will try to guess what your next character is. So as you type additional characters, it'll have a little drop down box which says, were you going to put this, were you going to put this? It will give you a little blurb that says, here's what I think that you were going to enter this next password.

So how does Telepathwords work? So it basically has a bunch of databases. It has a database of common passwords. It also has a list of popular phrases that it's taken from websites. And it also has this set of heuristics which describe common user biases in picking passwords.

So for example, one funny bias is that people will often-- when they are forced with these constraints to say you must use punctuation, stuff like that-- a lot of times when they're picking characters for the password, they will use keys that are adjacent to each other. So in other words, they'll be very small edit distance in physical space with respect to edit distance in the actual password.

So what a Telepathwords does is it has the database here, so when you type in things it's running these models. And it's saying, statistically speaking, here's the most likely thing that you're going to type next. So it's almost like auto complete for passwords.

And so what's funny is that this shows once again that if you have these constraints, they actually don't protect you that much if there are some of these underlying a priori distributions of things that the attacker can't leverage. I think there was a question?

AUDIENCE: Yeah so it seems like if an attacker is too sophisticated that they could try guessing like a bunch of IP addresses and things which only would prevent hammering [INAUDIBLE].

PROFESSOR: Yeah, it's very tricky. Now that's a good point. So anti-hammering basically sounds well what's the scope of the attack that you're trying to prevent? So if you're concerned about distributed attackers and a network system, it does become very, very subtle.

And suffice it to say that the notion of anti-hammering or [INAUDIBLE] systems, and also the notion of things like clipfraud, for example. So in other words, how does someone who's running an advertising campaign online determine if someone's actually putting the link and actually paying someone for those clicks, verses this is just spammer who got some box just sitting there clicking on stuff.

So suffice it to say there's a lot of distributed heuristics that try to solve those problems. And in many cases, it's not a science, it's an art. But your [INAUDIBLE] correct and in the distributed setting, things get much more difficult. All right, so does this all make sense?

AUDIENCE: What about the cryptographic anti-hammering defenses? Most of the time you end up sending a hash on the line [INAUDIBLE] that when you get out of it is exactly what you would get out the password of the hashable password? I know there are protocols like SRP or there are some zero knowledge protocols.

PROFESSOR: Yeah, so--

AUDIENCE: That you use in practice?

PROFESSOR: They do. Those protocols provides some stronger cryptographic guarantees. A lot of times they are not backwards compatible with current systems, which is why in practice you don't see them used a lot.

But yeah, there are some protocols, for example, that allow the server to not have any notion of the password at all. So there's some zero knowledge type thing or whatever. So those things do work in practice.

But one of the things that this paper says is very interesting is that you basically go through all these authentication schemes and they say, OK, here's passwords. Yeah, they kind of suck. Here's some other things that are actually much stronger on security access, but then they all fail on deployability or usability and things like that.

And so that's one of the interesting and slightly sad outcomes of this paper that maybe even though we have all these much stronger security for the protocols, we can't deploy them for some usability reasons or some [INAUDIBLE] reason.

So that's just a fun site to go to right. So they claim that they don't store your passwords so you take them at their word if you want to. But it is very interesting to just sit down and think like, what password I generate? And then type into this, and see how accurate it is in guessing what the next thing will be.

It even covers things like the popular heuristic like take a popular phrase that has multiple words, and then only take the first letter of each word. So this thing is very, very good. Very, very scary too. OK so that's Telepathwords.

And so one thing that is also interesting when you think about is in your password scheme, is it vulnerable to offline guessing. So this was a problem that Kerberos before that. And then also V5 without this thing they call preauth. So the basic idea is that in these versions of Kerberos, anyone could ask the KDC for a ticket that would encrypted with the users password.

So basically, the KDC did not authenticate requests that were coming from a client. Now the thing that the KDC would return was, in fact-- there are some set of bits here that the KDC would return. I'm sure you don't want to think about this ugly set of cryptographic printers anymore. But suffice it to say, the KDC would return this stuff that was encrypted with the key of the client. That's what will come back to the client side.

So the problem with this is that because the server did not check who was sending this encrypted set of things to, the attacker can basically get this thing here and then try to just guess what KC is. Just guess that KC is some value, try to encrypt this, see if it looks reasonable. If not, try to guess another KC, decrypt this, see if it looks reasonable.

And the reason why the attacker can launch this type of attack, is that this thing here, this TGT actually has a known format. So it has things in here like timestamps, and it has things in here like various link field would have to be internally consistent.

And so that basically helps the attacker. Because if the attacker guesses the KC, gets this thing here, a decrypted thing, and the internal fields don't check out, the attacker knows that it picked the wrong KC, so they can go on and pick another KC.

And so, in Kerberos V5, basically the client has to send in this thing that it sends over to the KDC, it basically sends a time stamp. And then this time stamp is going to be encrypted with KC. So this is sent to the server, and the server looks at this and validates that before it will send something back to the client. So that gets rid of this problem that any random client can show up and just ask for this thing here.

AUDIENCE: So is time stamp recorded in the message? So can't the attacker just give this message and enforce it?

PROFESSOR: Let's see here. So can't the attacker get this message here?

AUDIENCE: Yeah, the encryption [INAUDIBLE].

PROFESSOR: So you're thinking where the attacker might just spoof this, for example?

AUDIENCE: No, I just brute force it and get KC out.

PROFESSOR: OK. So in other words, you're worried someone could observe this.

AUDIENCE: Right.

PROFESSOR: So I believe that this is put inside an encrypted thing that belongs to the server, or the key belongs to the server. I think to prevent that attack. [INAUDIBLE] so don't quote me on that. But you're correct it's not, for example.

And if the attacker, for example, knew something that about what the current time is, roughly, that actually is super useful. Because then the attacker can guess, oh, time stamp should be roughly between here and here. And if it sees it's in the clear, it can do the exact same attack that we had up here.

AUDIENCE: It's a little better because the attacker has to be in the middle, but it's still susceptible.

PROFESSOR: That's true. Well, yeah, that's right, the attacker has to be on the network somewhere so this [INAUDIBLE] stuff. That's right. So that's all, I'm guessing.

So another thing that's important to think about is password recovery. So this is the idea that you lose your password, and then somehow you have to go to the service and you have to ask for another password. But before you get that password, you have to prove that you are you in some way.

So how does that work? How to do password recovery? So what's interesting is that people oftentimes focus on the entropy of the password itself. But the problem is that if the password recovery questions or the password recovery scheme has little entropy, that actually affects the entropy of the overall authentication scheme.

So in other words, the strength of the overall scheme is basically equal to the minimum of the password entropy in the recovery question entropy. And so you see this actually play out in a lot of rules scenarios.

There's a lot of famous cases, like the Sarah Palin case, where basically someone was able to recover her password fraudulently because her recovery questions were things that any random person could find. By looking at her Wikipedia article, for example, find out where she went to high school and things like that.

And so often times these password recovery questions are not very good. And they're not very good because of a couple reasons. So sometimes these things just have very low entropy. So if you have a password recovery question that is something like, what's your favorite color, the most popular answers are going to be like blue and red. Nobody's going to say like off white, fuchsia, magenta. So some of these recovery questions intrinsically are very difficult to provide a lot of entropy for.

The other problem is that sometimes these recover questions can be leaked via social media. So for example, if one of the recovery questions is what's your favorite movie? So maybe this space there is a little bit bigger, but if intrinsically I can go look at, let's say, your IMDB profile, your Facebook profile, and figure out like, oh hey, you literally told me that's your favorite movie, this isn't super useful either.

And another problem-- this is actually sort of the funniest one-- is that the user selected recovery questions are often super weak. So for example, people have done a survey of what some of these recovery questions look like, and sometimes users themselves will set recovery questions that are things like what is 2 plus 3?

And so, at the time, the user's thinking this is a big hassle, we're going to have to use this. But trivially most humans who pass the Turing Test can answer that questions successfully. And then therefore get the users password back.

AUDIENCE: So [INAUDIBLE] like using recovery passwords? It's basically like you enter in your name and maybe the subject of some emails that you've sent, like a small amount of additional information. But based on that, in some cases they can-- is security of that kind of stuff then?

PROFESSOR: So I don't know of any formal study like that. Those things are actually a lot better. I actually know this, because I was trying to help a friend go through this process. So she basically lost control of her Gmail account, and she was trying to prove that this was her account.

And so yeah, they would ask you things like roughly speaking, when did you open this account. Roughly speaking before you lost control of this account to hesball or whatever, who were some of the people that you talked to? And things like that. And it's actually a pretty laborious process. What ends up happening is that you're generally correct, it ends up being much more powerful than this stuff.

And so actually I don't know of any formal studies of that, but it does seem [INAUDIBLE] much strong than these types of things. All right, any other questions? Now we can get to the paper for today. So reading for today, the author has basically proposed a bunch of factors that can be used to evaluate these authentication schemes.

And what's really cool about this paper, I think, is that it basically tries to say, look, a lot of us in the security community are fighting just based on aesthetic principles. Like, we should pick this because I just like the way that the curly braces look in the proof. We should pick this because it uses a lot of math mode.

And so what they say is, look, why don't we try to establish some type of criteria? Maybe some of the criteria are a little bit subjective. Let's just try to have this taxonomy of ways to evaluate the authentication scheme. And let's just see how these various schemes stack up.

And so the authors basically proposed three high level metrics for evaluating these schemes. And so, the first metric is usability. And so, the base idea here is how easy is it for users to interact with this authentication scheme. So they find a couple interesting properties.

So for example, is it easy to learn? This basically just means is this scheme easy to learn? So some of these categories are pretty straightforward. Some of them actually involve a little bit of subtlety. But this one makes a lot of sense. And so if we look at passwords, passwords pass this test. Because everybody is used to using passwords, so we'll say they are easy to learn.

Another category is infrequent errors. So that means when you are trying to authenticate the system, if you are the actual user in question, is it the case that you can often authenticate yourself without generating errors?

And so, here the authors say quasi-yes. And so the quasi prefix is one of the more entertaining aspects of the paper, because authors kind of admit there's this element of subjectivity to it. So we can't necessarily say with crisp precision yes, no, things like this.

So the reason why they say quasi-yes is because, in general, you can authenticate a password successfully. But we've all been in that place where it's like 3 AM, we're trying to log on to our email server, our mind's not in the right place, and we enter a bunch of errors a bunch of times. So they say quasi-yes for this.

Another category is it scalable for users. And so the basic idea here is if the user has a bunch of different services that he or she wants to authenticate to, does this scheme scale well? Does the user have to remember some new thing for each one of the schemes?

And so, for here, the authors say no. Because in practice, it's very difficult for users to remember a separate password for every single site that they go to. This is one reason actually why people reuse their passwords often.

So another usability property is easy recovery. So what happens if you lose your authentication token-- in this case, your password-- is it easy to reset? And in this case, the answer for passwords is yes. In fact, they are probably too easy to reset, as we just discussed a couple minutes ago. So that's a yes.

And so another existing one is nothing to carry. So a lot of the more Barouque authentication protocols require you run some smartphone app, or you have some security token or smart card or things like that. So that's a burden.

Maybe not with a smartphone so much, but having to carry around one of these other gadgets is probably a pain. And so this is actually one nice feature of passwords, you basically only have to carry around in your brain, which is one that you should have at all moments.

So that's basically what usability looks like. It is very interesting in a high level that a lot of times these sort of factors are given a little bit of a short shrift in the community. Security can be when people are evaluating these schemes.

They say, oh, this thing uses like a million bits of entropy, and can only be broken by the Death Star or whatever. But then people don't necessarily remember these are actually very important factors too.

OK so the next high level category that the authors use to evaluate authentication scheme is deployability. So the base idea here is how easy is it to incorporate this system in to current web services.

So one thing they look at, for example, is is it server compatible? And this basically means can I easily integrate this scheme with today's servers, which are based around text based passwords?

And so since success here is defined with respect to passwords, passwords succeed. So another metric is browser compatibility. Similar type of thing. Can I use this scheme with current off-the-shelf browsers without having to install plug-in, something like that? Once again, passwords win by default.

And another interesting one is excessibility. So can people who can use passwords now, but maybe have some type of physical disability-- maybe they're blind, or they can't hear well, or they can't gesture well, or things like that. Can they actually use this scheme? This is actually pretty important.

So once again, the authors' saying yes. It's a little bit weird, because it's not clear that all people with all disabilities can use passwords, but they say yes here. So yes, so these are three interesting things to think about with respect to deployability. And the reason why this deployability category is so important is because it's very difficult to get anyone to upgrade anything ever.

I mean people don't even want to reboot their machines and get a new OS update installed. So it's very difficult that this scheme requires usable changes on the server to get people on the server to actually do different stuff. This goes back to your question, why don't we use these better things? Cause deployability in many cases is super, super important to people.

All right, so then the final category that we will look at is security. Right, so what kinds of attacks can this scheme prevent? So a lot of these security properties are resilient to foo. I'll just shorten that one of reds.

So is the scheme resilient to physical observations? So the idea here is that an attacker can not impersonate the user after observing them authenticate a few times. So imagine that you had a shoulder surfer.

So you're somewhere in a computer lab, someone's looking over your shoulder, seeing what you type in. Someone's videotaping you, maybe someone's got a microphone listening to the acoustic signature of your keyboard and trying to extract things from that, so on and so forth.

So the authors say that passwords actually failed this test. And that's because someone can videotape typing in things, they can pretty easily figure out what letters you typed. Or there's actually these attacks where you can actually listen to the acoustic fingerprint of the keyboard, and detect what was typed based on what sounds that you hear. So passwords are not resistant to physical observation.

So another property is resistant to targeted impersonation. And so the base idea here that, is that is it possible for someone who knows you-- a friend, an acquaintance, a spouse, a loved one, a family member, whatever-- to impersonate you using their knowledge of who you are and what you do.

So could your friend try to pretend to be you easily in this particular scheme? So here the authors basically have another one of these quasi-yeses. And they say quasi-yes because they're not aware of any studies which show that if you know a person, you're more likely to guess their password. So they say quasi-yes for that.

And so, note that resistance is targeted impersonation. This is where most security backup questions fail miserably. Because if someone knows something about you, quite easily they can guess your security questions in many cases.

So then we have two categories that involve guessing. So the first one is resilient to throttle guessing. And so what this means is if the attacker can not issue guesses at line rate, because for, example, the server uses anti-hammering mechanisms. Is the scheme safe against the attacker?

And so here, they say no. And so the reason why they say no, is because in practice passwords not only have sort of low inherit entropy because they're not that long, but also they have that skewed distribution.

And so what that means is that even if the attacker is throttled in some way, typically the attacker can still make good forward progress and crack a lot of people's passwords. So they define another guessing property which is resistant to unthrottled guessing.

And so this is basically saying, suppose that the attacker can issue these authentication forgery request as quickly as he or she wants. So in other words, the attacker is only limited by the speed of their hardware. So is the authentication scheme resilient to that type of attack?

And here maybe this answer's also no, for the same reason that the answer was no up here. So basically passwords have a very small entropy space and they come skewed distribution. So that's all pretty straightforward.

One interesting one is resiliency to internal observation. So this means that the attacker can not impersonate a user like intercepting that users input. For example, by installing a keystroke logger on the keyboard that the user's using, and using that logger to steal keypresses.

This also means, for example, that there's no way for network attacker who's observing the things that the client sending over the wire to use that knowledge of the network traffic to later impersonate the user.

And so here they say password do not have this scheme. And they essentially say it's because passwords are static tokens. They don't change. And typically static tokens are vulnerable to replay.

So if somehow, for example, an attacker installs a keystroke logger and gets your password, then basically the attacker can use that password until it's either expired or revoked or something that. It you just replay it again it'll go into that authenticating server on the other side. So here, passwords actually fail that test.

Another thing that we talked about a little bit in this class phishing. So resilience to phishing is another security metric. And the base idea here is that, if the attacker can simulate a valid service-- for example, by attacking the DNS infrastructure or something like that-- then the attacker cannot collect credentials from the user, then the attacker can then use to pretend to be the user later on.

And so this basically supposed penalized sites that do not strongly tell the user, hey, I'm this particular service, so you can feel confident to give me your credentials. And so if here passwords fail just because phishing sites are very, very popular. So passwords don't really intrinsically provide any protection against that.

Now the next two are particularly interesting in the context of a large scale distributed system. So no trusted third party. This essentially means that other than the client and the server, there's no one else in the system that is involved in the authentication protocol.

And so, that means that there's no third party who, if that third party were compromised, the entire integrity of the securities scheme might fall apart. And so, this is actually an interesting property to look at because a lot of authentication problems would go away if we could just store all our authentication information in one place.

We just store it in one place, it's very simple, we don't have to remember a lot of stuff on the client, we just say, whatever service you want to use, you always go to this one third party, and that third party will always be able to of authenticate you, and then allow you to go on your way.

Now of course third parties are problematic with perspective of robustness right because if you have one of these global third parties that everybody trusts, if that third party gets subverted then perhaps the integrity of all the sites that use that third party to authenticate all those sites are potentially in danger.

So they say that passwords do not have a trusted third party because each user is forced to have a separate password for each site. A related property is