6.1800 Spring 2024

Lecture #21: Authentication

authenticating users through the magic of (salted) hash functions

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

6.1800 in the news

NASA’s Voyager 1 Resumes Sending Engineering Updates to Earth

April 22, 2024

NASA’s Voyager 1 spacecraft is depicted in this artist’s concept traveling through interstellar space, or the space between stars, which it entered in 2012. Credit: NASA/JPL-Caltech

https://www.jpl.nasa.gov/news/nasas-voyager-1-resumes-sending-engineering-updates-to-earth

6.1800 in the news

fight data subsystem →

The team discovered that a single chip responsible for storing a portion of the FDS memory — including some of the FDS computer's software code — isn't working. The loss of that code rendered the science and engineering data unusable. Unable to repair the chip, the team decided to place the affected code elsewhere in the FDS memory. But no single location is large enough to hold the section of code in its entirety.

So they devised a plan to divide the affected code into sections and store those sections in different places in the FDS. To make this plan work, they also needed to adjust those code sections to ensure, for example, that they all still function as a whole. Any references to the location of that code in other parts of the FDS memory needed to be updated as well.

https://www.jpl.nasa.gov/news/nasas-voyager-1-resumes-sending-engineering-updates-to-earth

6.1800 in the news

The team started by singling out the code responsible for packaging the spacecraft's engineering data. They sent it to its new location in the FDS memory on April 18. A radio signal takes about 22 ½ hours to reach Voyager 1, which is over 15 billion miles (24 billion kilometers) from Earth, and another 22 ½ hours for a signal to come back to Earth. When the mission flight team heard back from the spacecraft on April 20, they saw that the modification worked: For the first time in five months, they have been able to check the health and status of the spacecraft.

https://www.jpl.nasa.gov/news/nasas-voyager-1-resumes-sending-engineering-updates-to-earth

6.1800 in the news

https://www.jpl.nasa.gov/news/nasas-voyager-1-resumes-sending-engineering-updates-to-earth

steps towards building a more secure system

1. be clear about goals (**policy**)
2. be clear about assumptions (**threat model**)

username	password (plaintext)
user1	fhxiyfioncvgxvrb
user2	pmhdtzyxjpwbsjwv
user3	gxrdqlbxrvhdopjg
user4	hncobhdvimrtbpmy
user5	jprfyxpznunmbabp
user6	abbsocpnkpnqirmf
user7	ybtsnxthnpunjrep
user8	eauohxeagxkejbyk
user9	viyigjfjdtllgdgd
user10	mwccvpbesqqkuwsu
user11	jpzxomdscorejjcb
user12	aqpyiegxyiftfpaa
user13	qopgbawviyrdzjhz
user14	ordkyoqkugmrzudj
user15	ygeblazpolvfufox
user16	bslrsuhqmsgfvfjx
user17	pwcdmcpucurnzmjy
user18	vlcfilrfcdjgtvqn
user19	jfkwdbgdprxnotuz
user20	cmhuajeqecvbdlyn
user21	tnfdfjiytbsomleb
user22	abpuvcbxqaoeaujq
user23	ykzpwiijhisbqlpw

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

to authenticate users:

```
check_password(username, inputted_password):
    stored_password = accounts_table[username]
    return stored_password == inputted_password
```

problem: the adversary (with access to the stored table) can just read the passwords directly

interlude: hash functions

hash functions are not normal functions! they have a number of exciting properties

a **hash function *H*** takes an input string of arbitrary size and outputs a fixed-length string

H is **deterministic**: if $x_1 = x_2$, then $H(x_1) = H(x_2)$

H is **collision-resistant**: if $x_1 \neq x_2$, then the probability that $H(x_1) = H(x_2)$ is virtually zero

H is **one-way**: given *x*, it is easy to compute *H(x)*. given *H(x)* (without *x*), it is virtually impossible to determine *x*

there are **very few** such functions, and the ones that exist are well-known

in the context of our quest for authentication, this means that we cannot rely on a “secret“ hash function; there is no such thing

we are going to use hash functions to (attempt to) solve our authentication problem

username	hash(password)
user1	14cdac7adb383312176cf4376d6bd594d4823e675567a545a0e13ada7f89c995
user2	adab20bca80250418a02196ca80d01d85456675b29816a40212eaa8d4c56ad22
user3	02c0d53273d41d74221c69c5393157965182254904aa0653aa43f3a206e6eaa1
user4	21f110ed053c24004463526b3ba06999c52580abeff8a85e89f7d9af966446e3
user5	2fae96a772f8e7c1bc5ae87ba291e4671b66792a024b324e87305ca551c34a80
user6	e5034d3265313a5b4fbac596b1e3f954805c0662c2f9cca6ca3efaa8282080cf
user7	5de31d1e08319d086c7fee8de384eb4157c6b02a6e575b7c49a6a9d8c7388836
user8	b8cd12c77b504b4e103dcc12cce948b2d737638d80c4e8c221cbbe13d1776bb7
user9	bbd108d7967db8489432dc3d1899d4d9c5d2f8d39ff106035d74757c140b527d
user10	1f7dbe35470e3f7364736afb9a02787c832ea8b5cdd958d53d27aa69c834e63b
user11	062f82f9ef701cfa2573aa40f2e7cc9879789512d55bafb2b72486ba262612d0
user12	224c37554776a15372b768a2c9232a57d27d29648e770deb91dadb4b6ddd5f3a
user13	92df155300535fa28fefe4a65f2059fb3bedfe6409d01cbf29dd45b1163234b4
user14	964205c227974d9048fe67a485f73f712d950dea64bf3c94cbf93fadec91d657
user15	907c14aae02712091b9682e5e08b997c69094d15b92e55b1463d247e10c9c235
user16	e30cff73a266cda4e41ad3e28d7770c6a029316de6da4c2976cba547b2efdb46
user17	6efa1445959a560e0148464ff4d316e94ae5d8f1948f6de483801fa12ed4056b

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

to authenticate users:

```
check_password(username, inputted_password):
  stored_hash = accounts_table[username]
  inputted_hash = hash(inputted_password)
  return stored_hash == inputted_hash
```

now the adversary cannot just read passwords directly from the table

H is **deterministic**: if $x_1 = x_2$, then $H(x_1) = H(x_2)$

H is **collision-resistant**: if $x_1 \neq x_2$, then the probability that $H(x_1) = H(x_2)$ is virtually zero

username	hash(password)
user1	14cdac7adb383312176cf4376d6bd594 d4823e675567a545a0e13ada7f89c995
user2	adab20bca80250418a02196ca80d01d8 5456675b29816a40212eaa8d4c56ad22
user3	02c0d53273d41d74221c69c539315796 5182254904aa0653aa43f3a206e6eaa1
user4	21f110ed053c24004463526b3ba06999 c52580abeff8a85e89f7d9af966446e3
user5	2fae96a772f8e7c1bc5ae87ba291e467 1b66792a024b324e87305ca551c34a80
user6	e5034d3265313a5b4fbac596b1e3f954 805c0662c2f9cca6ca3efaa8282080cf
user7	5de31d1e08319d086c7fee8de384eb41 57c6b02a6e575b7c49a6a9d8c7388836
user8	b8cd12c77b504b4e103dcc12cce948b2 d737638d80c4e8c221cbbe13d1776bb7
user9	bbd108d7967db8489432dc3d1899d4d9 c5d2f8d39ff106035d74757c140b527d
user10	1f7dbe35470e3f7364736afb9a02787c 832ea8b5cdd958d53d27aa69c834e63b
user11	062f82f9ef701cfa2573aa40f2e7cc98 79789512d55bafb2b72486ba262612d0
user12	224c37554776a15372b768a2c9232a57 d27d29648e770deb91dadb4b6ddd5f3a
user13	92df155300535fa28fefe4a65f2059fb 3bedfe6409d01cbf29dd45b1163234b4
user14	964205c227974d9048fe67a485f73f71 2d950dea64bf3c94cbf93fadec91d657
user15	907c14aae02712091b9682e5e08b997c 69094d15b92e55b1463d247e10c9c235
user16	e30cff73a266cda4e41ad3e28d7770c6 a029316de6da4c2976cba547b2efdb46
user17	6efa1445959a560e0148464ff4d316e9 4ae5d8f1948f6de483801fa12ed4056b

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

a **hash function *H*** takes an input string of arbitrary size and outputs a fixed-length string

H is **deterministic**: if $x_1 = x_2$, then $H(x_1) = H(x_2)$

H is **collision-resistant**: if $x_1 \neq x_2$, then the probability that $H(x_1) = H(x_2)$ is virtually zero

H is **one-way**: given *x*, it is easy to compute *H(x)*. given *H(x)* (without *x*), it is virtually impossible to determine *x*

there are **very few** such functions, and the ones that exist are well-known

question: suppose an adversary reads the hash of `user3`'s password. can the adversary figure out what the password was with that information alone?

username	hash(password)
user1	14cdac7adb383312176cf4376d6bd594 d4823e675567a545a0e13ada7f89c995
user2	adab20bca80250418a02196ca80d01d8 5456675b29816a40212eaa8d4c56ad22
user3	02c0d53273d41d74221c69c539315796 5182254904aa0653aa43f3a206e6eaa1
user4	21f110ed053c24004463526b3ba06999 c52580abeff8a85e89f7d9af966446e3
user5	2fae96a772f8e7c1bc5ae87ba291e467 1b66792a024b324e87305ca551c34a80
user6	e5034d3265313a5b4fbac596b1e3f954 805c0662c2f9cca6ca3efaa8282080cf
user7	5de31d1e08319d086c7fee8de384eb41 57c6b02a6e575b7c49a6a9d8c7388836
user8	b8cd12c77b504b4e103dcc12cce948b2 d737638d80c4e8c221cbbe13d1776bb7
user9	bbd108d7967db8489432dc3d1899d4d9 c5d2f8d39ff106035d74757c140b527d
user10	1f7dbe35470e3f7364736afb9a02787c 832ea8b5cdd958d53d27aa69c834e63b
user11	062f82f9ef701cfa2573aa40f2e7cc98 79789512d55bafb2b72486ba262612d0
user12	224c37554776a15372b768a2c9232a57 d27d29648e770deb91dadb4b6ddd5f3a
user13	92df155300535fa28fefe4a65f2059fb 3bedfe6409d01cbf29dd45b1163234b4
user14	964205c227974d9048fe67a485f73f71 2d950dea64bf3c94cbf93fadec91d657
user15	907c14aae02712091b9682e5e08b997c 69094d15b92e55b1463d247e10c9c235
user16	e30cff73a266cda4e41ad3e28d7770c6 a029316de6da4c2976cba547b2efdb46
user17	6efa1445959a560e0148464ff4d316e9 4ae5d8f1948f6de483801fa12ed4056b

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

to authenticate users:

```
check_password(username, inputted_password):
  stored_hash = accounts_table[username]
  inputted_hash = hash(inputted_password)
  return stored_hash == inputted_hash
```

now the adversary cannot just read passwords directly from the table, nor can they "reverse" the hash function to get the password

H is **one-way**: given *x*, it is easy to compute *H(x)*. given *H(x)* (without *x*), it is virtually impossible to determine *x*

H is **collision-resistant**: if $x_1 \neq x_2$, then the probability that $H(x_1) = H(x_2)$ is virtually zero

username	hash(password)
user1	14cdac7adb383312176cf4376d6bd594d4823e675567a545a0e13ada7f89c995
user2	adab20bca80250418a02196ca80d01d85456675b29816a40212eaa8d4c56ad22
user3	02c0d53273d41d74221c69c5393157965182254904aa0653aa43f3a206e6eaa1
user4	21f110ed053c24004463526b3ba06999c52580abeff8a85e89f7d9af966446e3
user5	2fae96a772f8e7c1bc5ae87ba291e4671b66792a024b324e87305ca551c34a80
user6	e5034d3265313a5b4fbac596b1e3f954805c0662c2f9cca6ca3efaa8282080cf
user7	5de31d1e08319d086c7fee8de384eb4157c6b02a6e575b7c49a6a9d8c7388836
user8	b8cd12c77b504b4e103dcc12cce948b2d737638d80c4e8c221cbbe13d1776bb7
user9	bbd108d7967db8489432dc3d1899d4d9c5d2f8d39ff106035d74757c140b527d
user10	1f7dbe35470e3f7364736afb9a02787c832ea8b5cdd958d53d27aa69c834e63b
user11	062f82f9ef701cfa2573aa40f2e7cc9879789512d55bafb2b72486ba262612d0
user12	224c37554776a15372b768a2c9232a57d27d29648e770deb91dadb4b6ddd5f3a
user13	92df155300535fa28fefe4a65f2059fb3bedfe6409d01cbf29dd45b1163234b4
user14	964205c227974d9048fe67a485f73f712d950dea64bf3c94cbf93fadec91d657
user15	907c14aae02712091b9682e5e08b997c69094d15b92e55b1463d247e10c9c235
user16	e30cff73a266cda4e41ad3e28d7770c6a029316de6da4c2976cba547b2efdb46
user17	6efa1445959a560e0148464ff4d316e94ae5d8f1948f6de483801fa12ed4056b

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

hash(password)	password
. . .	
14cdac7adb383312176cf4376d6bd594d4823e675567a545a0e13ada7f89c995	fhxiyfioncvgxvrb
adab20bca80250418a02196ca80d01d85456675b29816a40212eaa8d4c56ad22	pmhdtzyxjpwbsjwv
02c0d53273d41d74221c69c5393157965182254904aa0653aa43f3a206e6eaa1	gxrdqlbxrvhdopjg
21f110ed053c24004463526b3ba06999c52580abeff8a85e89f7d9af966446e3	hncobhdvimrtbpmy
2fae96a772f8e7c1bc5ae87ba291e4671b66792a024b324e87305ca551c34a80	jprfyxpznunmbabp
e5034d3265313a5b4fbac596b1e3f954805c0662c2f9cca6ca3efaa8282080cf	abbsocpnkpnqirmf
. . .	

it takes so little time to calculate a hash that it's feasible for the attacker to have the hash of *many* of our passwords

(it took my laptop 8 seconds to compute 10 million hashes)

an adversary can quickly make a table ahead of time, mapping passwords to their hashes, and use that to determine the user passwords

given x, it is easy to compute $H(x)$; given $H(x)$, it is virtually impossible to determine x.
there are **very few** such functions, and the ones that exist are well-known

interlude: hash functions

hash functions are not normal functions! they have a number of exciting properties

a **hash function *H*** takes an input string of arbitrary size and outputs a fixed-length string

H is **deterministic**: if $x_1 = x_2$, then $H(x_1) = H(x_2)$

H is **collision-resistant**: if $x_1 \neq x_2$, then the probability that $H(x_1) = H(x_2)$ is virtually zero

H is **one-way**: given *x*, it is easy to compute *H(x)*. given *H(x)* (without *x*), it is virtually impossible to determine *x*

there are **very few** such functions, and the ones that exist are well-known

in the context of our quest for authentication, this means that we cannot rely on a “secret“ hash function; there is no such thing

current problem: an adversary can easily pre-compute hashes for *a lot* of passwords

interlude: *slow* hash functions

hash functions are not normal functions! they have a number of exciting properties

a ***slow* hash function *H*** takes an input string of arbitrary size and outputs a fixed-length string

H is **deterministic**: if $x_1 = x_2$, then $H(x_1) = H(x_2)$

H is **collision-resistant**: if $x_1 \neq x_2$, then the probability that $H(x_1) = H(x_2)$ is virtually zero

H is **one-way**: given *x*, it is easy — **but slow** — to compute *H(x)*. given *H(x)* (without *x*), it is virtually impossible to determine *x*

there are **very few** such functions, and the ones that exist are well-known

in the context of our quest for authentication, this means that we cannot rely on a “secret“ hash function; there is no such thing

these slow hash functions are technically called “key derivation functions”, but we want to emphasize the relationship between them and the hash functions you’re already familiar with

they also have some additional properties, but not ones that we’re concerned with for this lecture

username	slow_hash(password)
user1	Vu7yYhTYngJSsXQARH/t0ExTjzMAvxm
user2	zg1MEP8gq.wSEEBs77IKSKpOfcPyVHi
user3	Q4echHFOG36HWzlr2PMBwD0mSBAnhF6
user4	1ea8JULlBbtIR01hKWnTj2m1CHuGvi6
user5	Dfhb0lYy2HSXsjbQ0tIT3FavbR5A3Sm
user6	EbsIBgo/birOFcP7sRpCXQGfxSfqX/G
user7	6KfAluC3xciYx/2McraVURKume2PSLG
user8	PW3jcT9uQEBRSppryJeAfq74AmFYk32
user9	NrYeWdAW641dHnWWagPG9FnzvyXkFQm
user10	thVVCSZCcZB3IbP.tRupSLKfseoSHKa
user11	dZTHFkjpzV29FcNDqmmwdhoF4exaKr.
user12	f90PAGN1eOh/neXJ9R71rj0saTGkFb2
user13	4AP5KW7YJS2HV699oDXKwz3nX1nAAK.
user14	7xhtnhapBhUEeG9HdJm1Ef5MlA9f/Nm
user15	dsgI4qRw98MQI1AbvVZZUynnQ/cs5Te
user16	D7LI3Qzct7fyBzvuXAXjbCz9a6UMYpm
user17	JMzSCQjujmgxXSg9.xFqC65Wz3C6RH6
user18	dINFGeda4wyIdtCqoGt7sI45Pd1rpYO
user19	UCshY6u.q83PKggZ5JuoaFHqHyb9R92
user20	KBvqt5KUmmmAHuZ525sw5ziYIOrIQ7q
user21	pIzJp/ypb1vsGEGRjUIcZBCYYGN1wcy
user22	A9CYWLRBPWcnguZt7wtBT0cunnrG1Ym
user23	Hm0noE1AIkTSEKm6t4sXmIKYU2xdAwS

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

to authenticate users:

```
check_password(username, inputted_password):
  stored_hash = accounts_table[username]
  inputted_hash = slow_hash(inputted_password)
  return stored_hash == inputted_hash
```

it would take my laptop about 23 days
to compute 10 million slow hashes,

given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x*. there are **very few** such functions, and the ones that exist are well-known

username	slow_hash(password)
user1	Vu7yYhTYngJSsXQARH/t0ExTjzMAvxm
user2	zg1MEP8gq.wSEEBs77IKSKpOfcPyVHi
user3	Q4echHFOG36HWzlr2PMBwD0mSBAnhF6
user4	1ea8JULlBbtIR01hKWnTj2m1CHuGvi6
user5	Dfhb0lYy2HSXsjbQ0tIT3FavbR5A3Sm
user6	EbsIBgo/birOFcP7sRpCXQGfxSfqX/G
user7	6KfAluC3xciYx/2McraVURKume2PSLG
user8	PW3jcT9uQEBRSppryJeAfq74AmFYk32
user9	NrYeWdAW641dHnWWagPG9FnzvyXkFQm
user10	thVVCSZCcZB3IbP.tRupSLKfseoSHKa
user11	dZTHFkjpzV29FcNDqmmwdhoF4exaKr.
user12	f90PAGN1eOh/neXJ9R71rj0saTGkFb2
user13	4AP5KW7YJS2HV699oDXKwz3nX1nAAK.
user14	7xhtnhapBhUEeG9HdJm1Ef5MlA9f/Nm
user15	dsgI4qRw98MQI1AbvVZZUynnQ/cs5Te
user16	D7LI3Qzct7fyBzvuXAXjbCz9a6UMYpm
user17	JMzSCQjujmgxXSg9.xFqC65Wz3C6RH6
user18	dINFGeda4wyIdtCqoGt7sI45Pd1rpYO
user19	UCshY6u.q83PKggZ5JuoaFHqHyb9R92
user20	KBvqt5KUmmmAHuZ525sw5ziYIOrIQ7q
user21	pIzJp/ypb1vsGEGRjUIcZBCYYGN1wcy
user22	A9CYWLRBPWcnguZt7wtBT0cunnrG1Ym
user23	Hm0noE1AIkTSEKm6t4sXmIKYU2xdAwS

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

slow_hash(password)	password
. . .	
	fhxiyfioncvgxvrb
	pmhdtzyxjpwbsjwv
	gxrdqlbxrvhdopjg
	hncobhdvimrtbpmy
	jprfyxpznunmbabp
	abbsocpnkpnqirmf
. . .	

is it feasible for an adversary to make the same table as before?

it is for fewer passwords!

given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x.* there are **very few** such functions, and the ones that exist are well-known

6.1800 in the news

RANK	PASSWORD	TIME TO CRACK IT	COUNT
1	**123456**	< 1 Second	4,524,867
2	**admin**	< 1 Second	4,008,850
3	**12345678**	< 1 Second	1,371,152
4	**123456789**	< 1 Second	1,213,047
5	**1234**	< 1 Second	969,811
6	**12345**	< 1 Second	728,414
7	**password**	< 1 Second	710,321
8	**123**	< 1 Second	528,086
9	**Aa123456**	< 1 Second	319,725
10	**1234567890**	< 1 Second	302,709

https://nordpass.com/most-common-passwords-list/

username	slow_hash(password)
user1	Vu7yYhTYngJSsXQARH/t0ExTjzMAvxm
user2	5OTqbEyfWGlI95NA9zQFTrqte6c1y5q
user3	T7bMFpBeUqOM09QRbYE6l1hAOQcvVXy
user4	1ea8JULlBbtIR01hKWnTj2m1CHuGvi6
user5	5OTqbEyfWGlI95NA9zQFTrqte6c1y5q
user6	EbsIBgo/birOFcP7sRpCXQGfxSfqX/G
user7	6KfAluC3xciYx/2McraVURKume2PSLG
user8	PW3jcT9uQEBRSppryJeAfq74AmFYk32
user9	NrYeWdAW641dHnWWagPG9FnzvyXkFQm
user10	n3CrTA6Tk5a0F/oHioWesdj1Zeirhla
user11	dZTHFkjpzV29FcNDqmmwdhoF4exaKr.
user12	f90PAGN1eOh/neXJ9R71rj0saTGkFb2
user13	n3CrTA6Tk5a0F/oHioWesdj1Zeirhla
user14	7xhtnhapBhUEeG9HdJm1Ef5MlA9f/Nm
user15	dsgI4qRw98MQI1AbvVZZUynnQ/cs5Te
user16	D7LI3Qzct7fyBzvuXAXjbCz9a6UMYpm
user17	JMzSCQjujmgxXSg9.xFqC65Wz3C6RH6
user18	P8.czpjwjtTR5aLdN7lhP17mbOZ.He2
user19	UCshY6u.q83PKggZ5JuoaFHqHyb9R92
user20	KBvqt5KUmmmAHuZ525sw5ziYIOrIQ7q
user21	pIzJp/ypb1vsGEGRjUIcZBCYYGN1wcy
user22	sEh47bf/w44wPdoHp5y0qWNabaVfa.G
user23	Hm0noE1AIkTSEKm6t4sXmIKYU2xdAwS

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

slow_hash(password)	password
. . .	
5OTqbEyfWGlI95NA9zQFTrqte6c1y5q	111111
n3CrTA6Tk5a0F/oHioWesdj1Zeirhla	password
T7bMFpBeUqOM09QRbYE6l1hAOQcvVXy	123456
K.V0OUcnrbljbFlRsQksDzrXHrNkwYG	asdfasdf
P8.czpjwjtTR5aLdN7lhP17mbOZ.He2	123123
sEh47bf/w44wPdoHp5y0qWNabaVfa.G	qwerty
. . .	

because it takes much longer to make this table using slow hashes, the attacker is incentivized to concentrate on the most common passwords

is it feasible for an adversary to make the same table as before?

it is for fewer passwords!

given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x*. there are **very few** such functions, and the ones that exist are well-known

highlighted users are using common passwords

username	slow_hash(password)
user1	Vu7yYhTYngJSsXQARH/t0ExTjzMAvxm
user2	5OTqbEyfWGlI95NA9zQFTrqte6c1y5q
user3	T7bMFpBeUqOM09QRbYE6l1hAOQcvVXy
user4	1ea8JULlBbtIR01hKWnTj2m1CHuGvi6
user5	5OTqbEyfWGlI95NA9zQFTrqte6c1y5q
user6	EbsIBgo/birOFcP7sRpCXQGfxSfqX/G
user7	6KfAluC3xciYx/2McraVURKume2PSLG
user8	PW3jcT9uQEBRSppryJeAfq74AmFYk32
user9	NrYeWdAW641dHnWWagPG9FnzvyXkFQm
user10	n3CrTA6Tk5a0F/oHioWesdj1Zeirhla
user11	dZTHFkjpzV29FcNDqmmwdhoF4exaKr.
user12	f90PAGN1eOh/neXJ9R71rj0saTGkFb2
user13	n3CrTA6Tk5a0F/oHioWesdj1Zeirhla
user14	7xhtnhapBhUEeG9HdJm1Ef5MlA9f/Nm
user15	dsgI4qRw98MQI1AbvVZZUynnQ/cs5Te
user16	D7LI3Qzct7fyBzvuXAXjbCz9a6UMYpm
user17	JMzSCQjujmgxXSg9.xFqC65Wz3C6RH6
user18	P8.czpjwjtTR5aLdN7lhP17mbOZ.He2
user19	UCshY6u.q83PKggZ5JuoaFHqHyb9R92
user20	KBvqt5KUmmmAHuZ525sw5ziYIOrIQ7q
user21	pIzJp/ypb1vsGEGRjUIcZBCYYGN1wcy
user22	sEh47bf/w44wPdoHp5y0qWNabaVfa.G
user23	Hm0noE1AIkTSEKm6t4sXmIKYU2xdAwS

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

we would like to make it more difficult for adversaries to pre-compute hashes, even for common passwords

idea: add randomness

given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x*. there are **very few** such functions, and the ones that exist are well-known

highlighted users are using common passwords

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

we're going to associate a random value — a *salt* — with each user. these salts are stored in plaintext; they are not a secret

instead of storing a hash of the password, we will concatenate the password and the salt, and store the hash of that string

given x, it is easy but slow to compute $H(x)$; given $H(x)$, it is virtually impossible to determine x. there are **very few** such functions, and the ones that exist are well-known

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

question: how do we authenticate users in this system? I.e., how does the below code need to change?

```
check_password(username, inputted_password):
  stored_hash = accounts_table[username]
  inputted_hash = slow_hash(inputted_password)
  return stored_hash == inputted_hash
```

this is our previous `check_password` function

given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x*. there are **very few** such functions, and the ones that exist are well-known

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

to authenticate users:

```
check_password(username, inputted_password)
  stored_hash = accounts_table[username].hash
  salt = accounts_table[username].salt
  inputted_hash = slow_hash(inputted_password | salt)
  return stored_hash == inputted_hash
```

given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x*. there are **very few** such functions, and the ones that exist are well-known

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

slow_hash(password|salt)	password
. . .	
?	111111
?	123456
?	password
. . .	

is it feasible for an adversary to make the same table as before?

to pre-compute the table, even for just the most common passwords, the adversary would need a separate table for *every possible salt*; this is infeasible

to target a specific user — say `user1` — an adversary could read `user1`'s salt and make a table using that salt. the more common `user1`'s password, the more likely it would be revealed, but it's unlikely an adversary would pre-compute this table since they would not know the salt ahead of time.

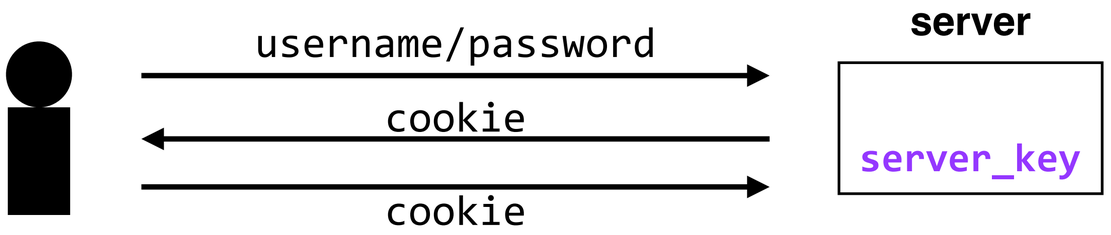
given *x*, it is easy but slow to compute *H(x)*; given *H(x)*, it is virtually impossible to determine *x*. there are **very few** such functions, and the ones that exist are well-known

limiting transmission of passwords

we want users to transmit passwords infrequently, because transmitting a password repeatedly can open a user up to other attacks outside of our current threat model

cookies allow users to authenticate themselves from some period of time, without repeatedly retransmitting their passwords

threat model: adversary can observe the cookie and is trying to learn the user's password, or to learn enough information such that it could create cookies that were valid for a long period of time

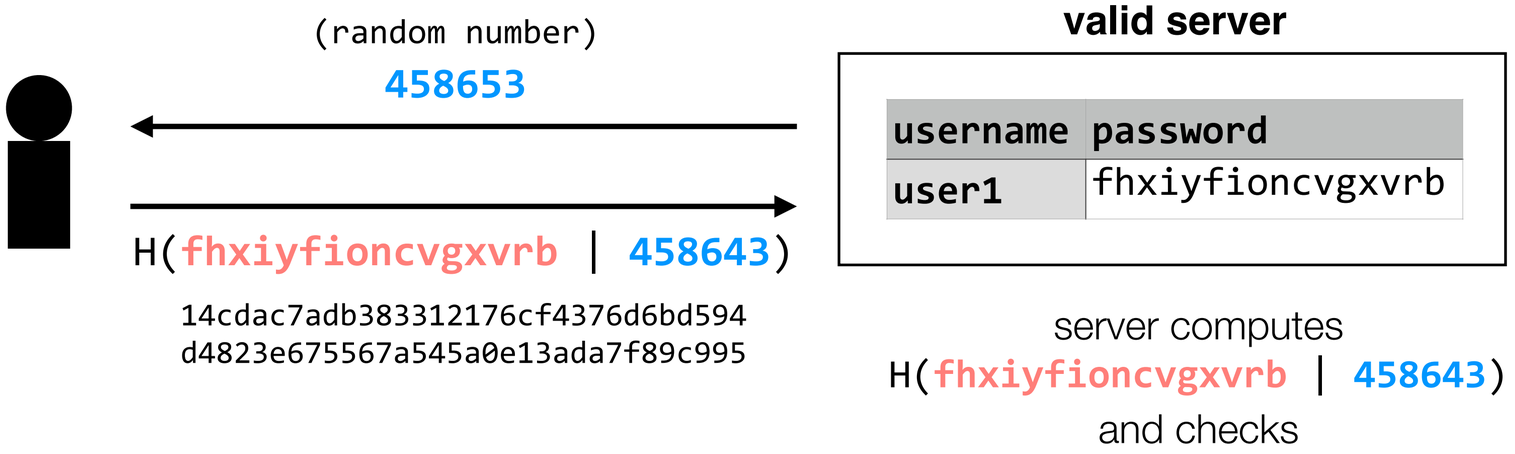


`cookie = {user, expiration, H(server_key | user | expiration)}`

`user` here is more than just a username; it also contains information about the user's browser

challenge-response protocols authenticate users without ever transmitting the password, and can help combat phishing attacks

threat model: adversary owns a server, and is trying to convince the user to send their password directly to the adversary

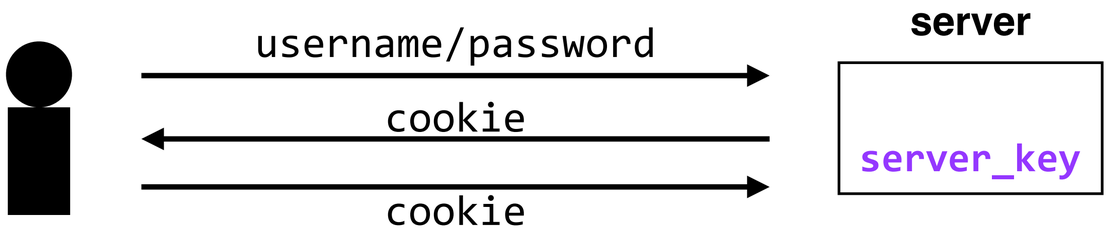


limiting transmission of passwords

we want users to transmit passwords infrequently, because transmitting a password repeatedly can open a user up to other attacks outside of our current threat model

cookies allow users to authenticate themselves from some period of time, without repeatedly retransmitting their passwords

threat model: adversary can observe the cookie and is trying to learn the user's password, or to learn enough information such that it could create cookies that were valid for a long period of time

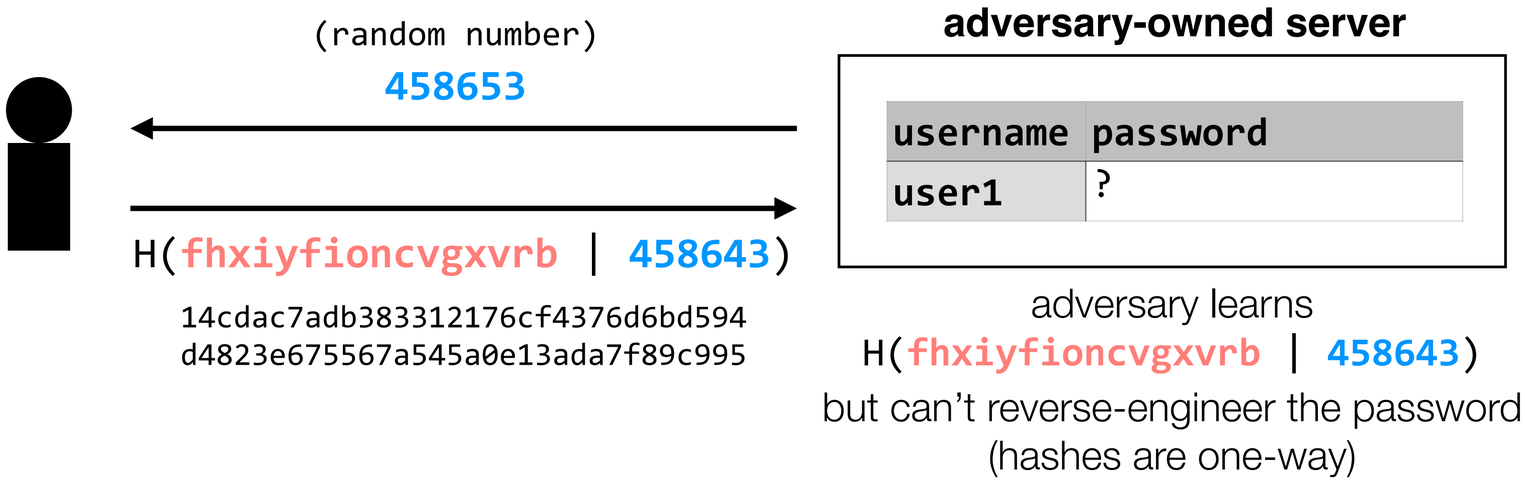


`cookie = {user, expiration, H(server_key | user | expiration)}`

`user` here is more than just a username; it also contains information about the user's browser

challenge-response protocols authenticate users without ever transmitting the password, and can help combat phishing attacks

threat model: adversary owns a server, and is trying to convince the user to send their password directly to the adversary



adversary learns
H(fhxiyfioncvgxvrb | 458643)
but can't reverse-engineer the password
(hashes are one-way)

in this specific setup, problems arise when the valid server is using salted hashes (as it should be), but challenge-response protocols exist to handle that case

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

are there alternatives to passwords?

yes, and they have their own trade-offs

using passwords to provide **authentication** takes some effort. storing **salted hashes**, incorporating **session cookies** and **challenge-response protocols** can help mitigate common attacks

passwords provide more **general lessons** about security: consider human factors when designing secure systems, in particular

there are always **trade-offs**. many proposed improvements on passwords do add security, but often add complexity and decrease usability