Scriptless Bitcoin Lotteries from Oblivious Transfer

Lloyd Fournier

lloyd.fourn@gmail.com

September 11, 2019

Rust is good for crypto

1. Rust is low level where it counts
2. It has nice saftey features
3. It has high quality crypto libraries (that are always getting better)
4. You can expose your crypto libraries using FFI
5. You can compile to web assembly!

Why you should never roll your own crypto

There are two meanings to the statement:

1. Don't design your own cryptographic protocols.
2. Don't implement cryptographic protocols that other people designed.

cryptographic protocols are notoriously difficult to design and implement.
You should only use cryptography that has security proofs and has been peer reviewed.
Only use crypto libraries in production that have been extensively vetted

Why you should never roll your own crypto

There are two meanings to the statement:

1. Don't design your own cryptographic protocols.
2. Don't implement cryptographic protocols that other people designed.

cryptographic protocols are notoriously difficult to design and implement.
You should only use cryptography that has security proofs and has been peer reviewed.
Only use crypto libraries in production that have been extensively vetted

(Unless you are making cryptocurrency then just roll with whatever you've got)

Why you should never roll your own crypto

There are two meanings to the statement:

1. Don't design your own cryptographic protocols.
2. Don't implement cryptographic protocols that other people designed.

cryptographic protocols are notoriously difficult to design and implement.
You should only use cryptography that has security proofs and has been peer reviewed.
Only use crypto libraries in production that have been extensively vetted

(Unless you are making cryptocurrency then just roll with whatever you've got)

(We will do both today)

But but but...

1. What if the existing cryptographic protocols don't do what you want?
2. What if there's not implementation of the cryptogrpahic protocol you want?

Well let's make an exception

There is an exception to this rule.
What if whatever you're already doing is so bad that it's hard to make it worse and there is no existing crypto to make it better.

Well let's make an exception

There is an exception to this rule.
What if whatever you're already doing is so bad that it's hard to make it worse and there is no existing crypto to make it better.

For example, You are making your users send their passwords to your server when they log in.

Well let's make an exception

There is an exception to this rule.
What if whatever you're already doing is so bad that it's hard to make it worse and there is no existing crypto to make it better.

For example, You are making your users send their passwords to your server when they log in.

(hint: You are)

Warning

Nothing in this talk should be taken as advice on security or cryptoraphy.

Warning

Nothing in this talk should be taken as advice on security or cryptoraphy.
This talk is for entertainment purposes only.

How can we judge how bad a password based login system is? Two main types of adversarial scenarios:

1. A breached database
2. A semi-honest server

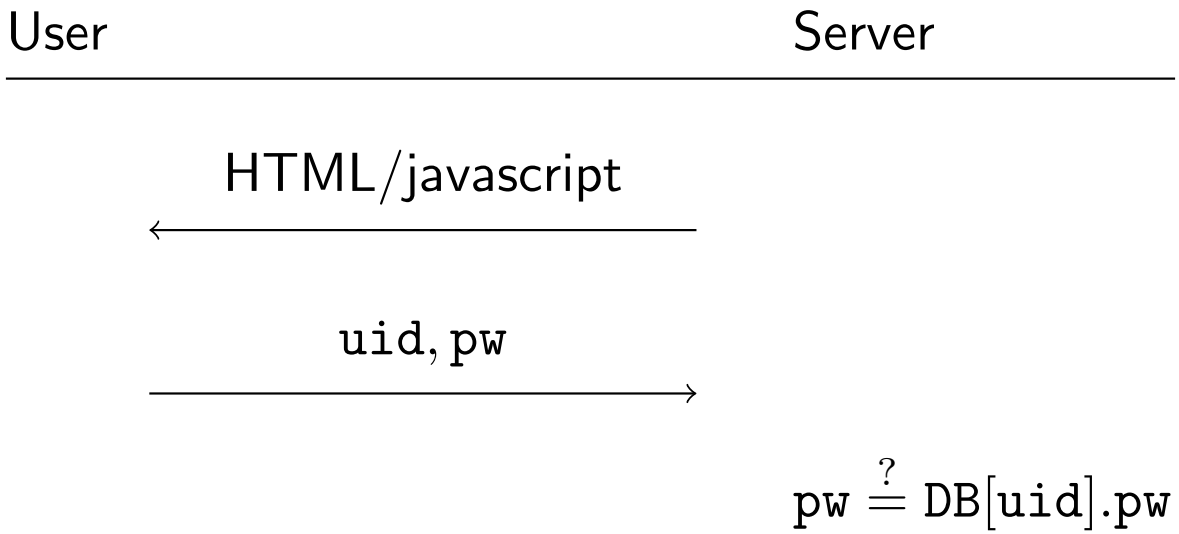
How can we judge how bad a password based login system is? Two main types of adversarial scenarios:

1. A breached database
2. A semi-honest server

Most people only think about (1) and completely ignore (2).

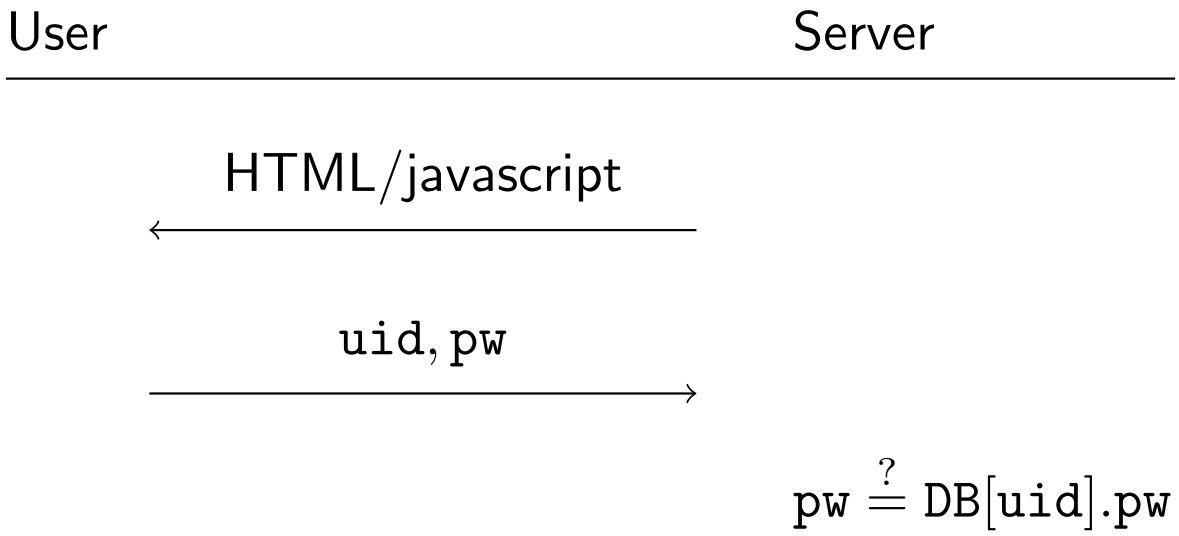
Security against semi-honest server

Semi-honest adversaries follow the protocol (they won't send a malicious HTML or JS) but they will try and learn stuff from the messages sent.



Security against semi-honest server

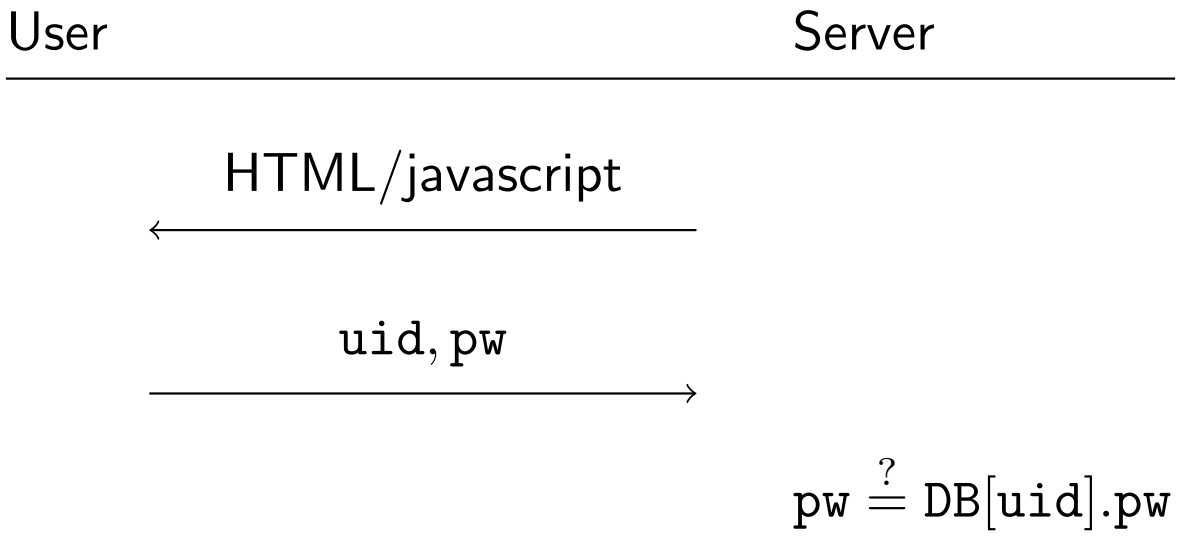
Semi-honest adversaries follow the protocol (they won't send a malicious HTML or JS) but they will try and learn stuff from the messages sent.



Imagine that the server does what it's meant to but then later leaks the transcript to the worst guy in the world.

Security against semi-honest server

Semi-honest adversaries follow the protocol (they won't send a malicious HTML or JS) but they will try and learn stuff from the messages sent.



Imagine that the server does what it's meant to but then later leaks the transcript to the worst guy in the world.
Is this realistic?

Yes.

1. March 2019 - Facebook reveals hundreds of millions of Facebook Lite user's passwords were stored in plaintext accidentally (and millions of instagram) *since 2012*.
2. May 2018 - Twitter and Github reveals that all passwords were being logged before being hashed

Yes.

1. March 2019 - Facebook reveals hundreds of millions of Facebook Lite user's passwords were stored in plaintext accidentally (and millions of instagram) *since 2012*.
2. May 2018 - Twitter and Github reveals that all passwords were being logged before being hashed

How many companies realise this and do not tell anyone? How many don't realise it? How many times have these passwords been used maliciously?

Login version 0.1 - Client side hashing

User		Server
	$(\text{sign-up}, \texttt{uid}, \texttt{pw}) \longrightarrow$	$\texttt{DB}[\texttt{uid}].\texttt{salt} \leftarrow \text{random}()$
		$\texttt{DB}[\texttt{uid}].\texttt{hash} \leftarrow \text{scrypt}(\texttt{pw}, \texttt{DB}[\texttt{uid}].\texttt{salt})$
	$(\text{login}, \texttt{uid}, \texttt{pw}) \longrightarrow$	
		$\texttt{hash} \leftarrow \text{scrypt}(\texttt{pw}, \texttt{DB}[\texttt{uid}].\texttt{salt})$
		$\texttt{hash} \stackrel{?}{=} \texttt{DB}[\texttt{uid}].\texttt{hash}$

Login version 0.1 - Client side hashing

User		Server
	$(\text{sign-up}, \texttt{uid}, \texttt{pw}) \longrightarrow$	$\texttt{DB}[\texttt{uid}].\texttt{salt} \leftarrow \text{random}()$
		$\texttt{DB}[\texttt{uid}].\texttt{hash} \leftarrow \text{scrypt}(\texttt{pw}, \texttt{DB}[\texttt{uid}].\texttt{salt})$
	$(\text{login}, \texttt{uid}, \texttt{pw}) \longrightarrow$	
		$\texttt{hash} \leftarrow \text{scrypt}(\texttt{pw}, \texttt{DB}[\texttt{uid}].\texttt{salt})$
		$\texttt{hash} \stackrel{?}{=} \texttt{DB}[\texttt{uid}].\texttt{hash}$

Data breach is protected but adversary still learns password.

Login version 0.2 - Client side hashing

User		Server$^{\mathcal{A}}$
$\texttt{salt}_c \leftarrow$ "domain.com"$\|\|\texttt{uid}$		
$\texttt{hash}_c \leftarrow \text{scrypt}(\texttt{pw}, \texttt{salt}_c)$		
	$(\text{sign-up}, \texttt{uid}, \texttt{hash}_c) \longrightarrow$	$\texttt{DB}[\texttt{uid}].\texttt{salt} \leftarrow \text{random}()$
		$\texttt{DB}[\texttt{uid}].\texttt{hash} \leftarrow \text{scrypt}(\texttt{hash}_c, \texttt{DB}[\texttt{uid}].\texttt{salt})$
	$(\text{login}, \texttt{uid}, \texttt{hash}_c) \longrightarrow$	
		$\texttt{hash} \leftarrow \text{scrypt}(\texttt{hash}_c, \texttt{DB}[\texttt{uid}].\texttt{salt})$
		$\texttt{hash} \stackrel{?}{=} \texttt{DB}[\texttt{uid}].\texttt{hash}$

Login version 0.2 - Client side hashing

User		Server$^{\mathcal{A}}$
$\texttt{salt}_c \leftarrow$ "domain.com"$\|\|\texttt{uid}$		
$\texttt{hash}_c \leftarrow \text{scrypt}(\texttt{pw}, \texttt{salt}_c)$		
	$(\text{sign-up}, \texttt{uid}, \texttt{hash}_c) \longrightarrow$	$\texttt{DB}[\texttt{uid}].\texttt{salt} \leftarrow \text{random}()$
		$\texttt{DB}[\texttt{uid}].\texttt{hash} \leftarrow \text{scrypt}(\texttt{hash}_c, \texttt{DB}[\texttt{uid}].\texttt{salt})$
	$(\text{login}, \texttt{uid}, \texttt{hash}_c) \longrightarrow$	
		$\texttt{hash} \leftarrow \text{scrypt}(\texttt{hash}_c, \texttt{DB}[\texttt{uid}].\texttt{salt})$
		$\texttt{hash} \stackrel{?}{=} \texttt{DB}[\texttt{uid}].\texttt{hash}$

We stopped $\mathcal{A}$ from learning our password and trying it on other websites.

Login version 0.2 - Client side hashing

User		Server$^{\mathcal{A}}$
$\texttt{salt}_c \leftarrow$ "domain.com"$\|\|\texttt{uid}$		
$\texttt{hash}_c \leftarrow \text{scrypt}(\texttt{pw}, \texttt{salt}_c)$		
	$(\text{sign-up}, \texttt{uid}, \texttt{hash}_c) \longrightarrow$	$\texttt{DB}[\texttt{uid}].\texttt{salt} \leftarrow \text{random}()$
		$\texttt{DB}[\texttt{uid}].\texttt{hash} \leftarrow \text{scrypt}(\texttt{hash}_c, \texttt{DB}[\texttt{uid}].\texttt{salt})$
	$(\text{login}, \texttt{uid}, \texttt{hash}_c) \longrightarrow$	
		$\texttt{hash} \leftarrow \text{scrypt}(\texttt{hash}_c, \texttt{DB}[\texttt{uid}].\texttt{salt})$
		$\texttt{hash} \stackrel{?}{=} \texttt{DB}[\texttt{uid}].\texttt{hash}$

We stopped $\mathcal{A}$ from learning our password and trying it on other websites. $\mathcal{A}$ can still login as anyone in our system though. $\mathcal{A}$ can also offline attack $\texttt{hash}_c$ to try and figure out our password.

Google on client side hashing

Have the client computer hash the password using a cryptographically secure algorithm and a unique salt provided by the server. When the password is received by the server, hash it again with a different salt that is unknown to the client. Be sure to store both salts securely. If you are using a modern and secure hashing algorithm, repeated hashing does not reduce entropy.

— Google, Modern password security for system designers

Google on client side hashing

> *Have the client computer hash the password using a cryptographically secure algorithm and a unique salt provided by the server. When the password is received by the server, hash it again with a different salt that is unknown to the client. Be sure to store both salts securely. If you are using a modern and secure hashing algorithm, repeated hashing does not reduce entropy.*
>
> — Google, Modern password security for system designers

Wrong. There is not point having the server provide the client salt. The attacker can just ask the server for the salt of any user so it is not secret even if database is not breached. If not extremely careful this can be used to check whether the user exists.

Introducing DRYUP

Don't **R**eveal **Y**our **U**ser's **P**asswords.

Diffie-Hellman Key Exchange

P_1		P_2
$a \leftarrow_{\$} \mathbb{Z}_q$		$b \leftarrow_{\$} \mathbb{Z}_q$
$A \leftarrow g^a$		$B \leftarrow g^b$
	$A \longrightarrow$	
	$\longleftarrow B$	
$key \leftarrow B^a$		$key \leftarrow A^b$

$$key = B^a = A^b = g^{ab}$$

You can't compute key from A and B: *Computational Diffie-Hellman assumption* (CDH)

As an authentication protocol

User(a)		Server
		$b \leftarrow_{\$} \mathbb{Z}_q$
$A \leftarrow g^a$		$B \leftarrow g^b$
	$B \longleftarrow$	
$key \leftarrow B^a$		
	$A, key \longrightarrow$	$key \stackrel{?}{=} A^b$

You can't produce (g^a, g^{ab}) given random g^b unless you know a:
Knowledge of Exponent Assumption (KEA1) (NOT the same as CDH).

User		Server$^{\mathcal{A}}$
$\texttt{salt}_c \leftarrow$ "domain.com"$\,||\texttt{uid}$		
$a \leftarrow \text{scrypt}(\texttt{pw}, \texttt{salt}_c)$		
$g \leftarrow \text{hash\_to\_point}(\text{"domain.com"})$		
$A \leftarrow g^a$		
	$\xrightarrow{(\text{sign-up}, \texttt{uid}, A)}$	$\texttt{DB}[\texttt{uid}].\texttt{salt} \leftarrow \text{random}()$
		$\texttt{DB}[\texttt{uid}].\texttt{hash} \leftarrow \text{scrypt}(A, \texttt{DB}[\texttt{uid}].\texttt{salt})$
	$\xleftarrow{B}$	$b \leftarrow_{\$} \mathbb{Z}_q; B \leftarrow g^b$
$(a, A) \leftarrow$		
$key \leftarrow B^a$		
	$\xrightarrow{(\text{login}, \texttt{uid}, A, key)}$	
		$\texttt{hash} \leftarrow \text{scrypt}(A, \texttt{DB}[\texttt{uid}].\texttt{salt})$
		$\texttt{hash} \stackrel{?}{=} \texttt{DB}[\texttt{uid}].\texttt{hash}$
		$key \stackrel{?}{=} A^b$

Implement in Rust!

- For public key cryptography we use *ristretto* `dalek-cryptography/curve25519-dalek`
- For scrypt we use one from `https://github.com/RustCrypto`
- And compile them to WASM for the client

So why does no one do this?

Some people actually do: https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/. Apple actually uses a similar scheme for iCloud Key Vault.
In academia, these schemes are referred to as Password Authenticated Key Exchange (PAKE). Quite a lot of research has gone into them. I haven't dived into it yet to confidently say how the scheme I came up with compares to them.

1. It is unlikely to improve the profitability of your business
2. It is an improvement of the security for users rather than business
3. Most unauthorised access will be caused by phishing and bad passwords.
4. cryptographers are bad at explaining what crypto can do!

The End