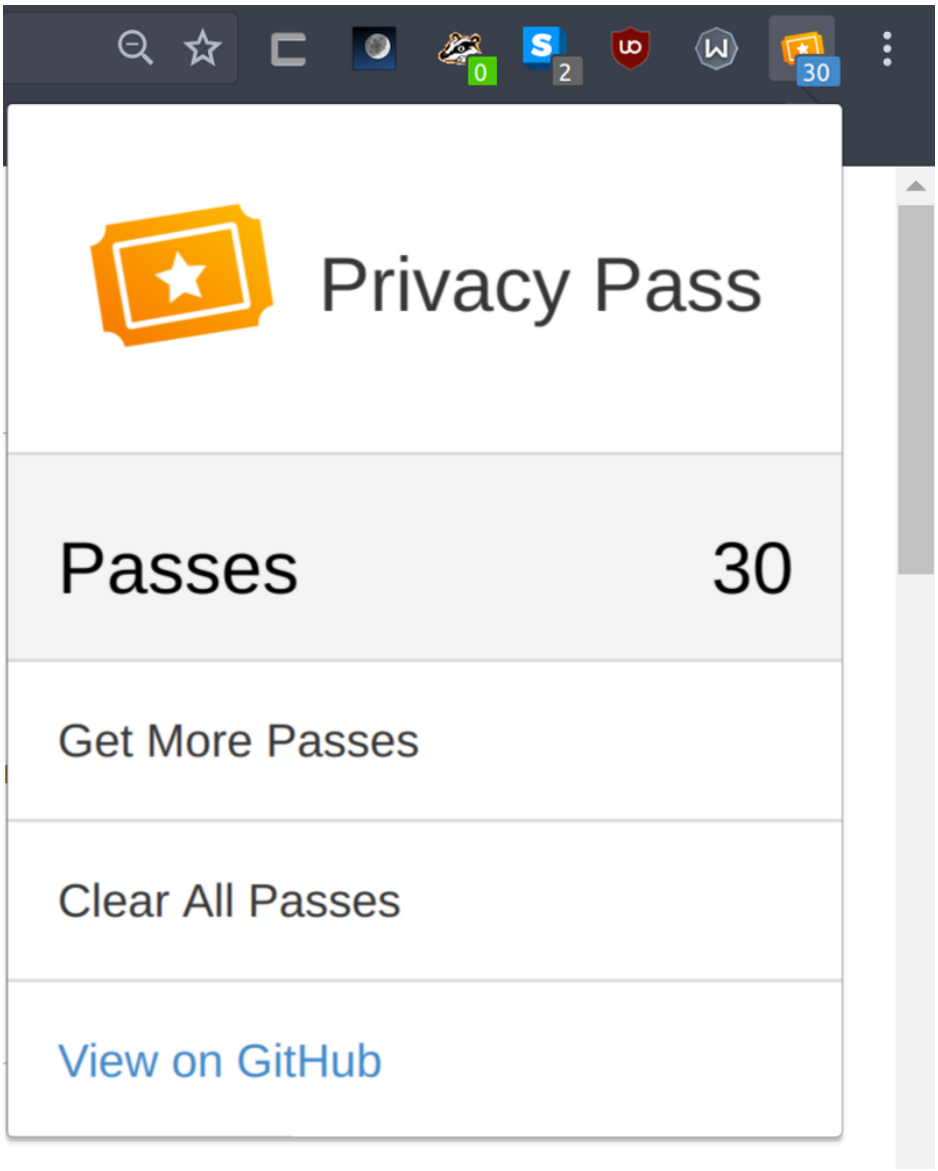
Alex Davidson Follow
Nov 9 · 11 min read

Privacy Pass: A browser extension for anonymous authentication

Today we are releasing a new browser extension for Chrome and Firefox named Privacy Pass. Privacy Pass uses privacy-preserving cryptography to allow users to authenticate to services without compromising their anonymity. This is joint work with George Tankersley, Ian Goldberg, Nick Sullivan and Filippo Valsorda.



Summary

Privacy Pass aims to make the internet more accessible for users with strict anonymity constraints. Authentication is usually handled via performing a log in for each of the different services that a user interacts with. However, there are some services where a log in is not possible, or where identifying a client that has performed some task in the past may be useful. For example, a client may be granted access to a resource if it has completed some sort of proof-of-work.

Notionally, we could achieve such a system by just handing tokens to clients that can then be redeemed in the future (much like a cookie). Unfortunately, if a service controls both the dissemination and consumption of these tokens then they can track the users. For example, even if a user is using some sort of tool that ensures some level of anonymity (e.g. using I2P, VPNs, Tor) then these cookies could be used to link the browsing patterns of disparate sessions.

Our aim with Privacy Pass is to create an authentication mechanism that does not compromise the anonymity of the user. Using the extension, users can retrieve 'passes' from services by completing a task of the server's choosing. However, when the user comes to redeem a pass in the future, it will be cryptographically unlinkable from the pass that was received before; even for the service that issued it. We achieve this using a cryptographic protocol based on a concept known as a '*Verifiable Oblivious Pseudorandom Function*' (VOPRF). Additionally, we use cryptographic 'blinding', originally introduced by Chaum for blind signature schemes, to construct a protocol that maintains *unlinkability*.

In short, our protocol first allows a service to sign blinded tokens generated client-side by the browser extension. These blinded tokens are sent to services along with some valid authentication method (e.g. challenge solution). The tokens are signed by the server and returned to the client; Privacy Pass 'unblinds' the returned tokens and stores them for future redemptions. When a client is required to perform another authentication for the same service, Privacy Pass constructs a pass from an unspent token and sends it instead. The service verifies that the token was signed previously and then provides the client with access.

Since the blind is randomly generated by the client and never seen by the service, we ensure that the service cannot link a token that was signed to an unblinded token that is redeemed later. We detail the protocol more specifically below. Our protocol is constructed using elliptic curve operations (we use the NIST P-256 curve) and thus the additional cryptographic overhead is minimal. We envisage that Privacy Pass could be used with multiple different services at the same time even if services use different private keys for signing.

Privacy Pass is released as beta currently, but we hope that releasing it now will encourage more users to get involved in the development process.

The Protocol

Our usage of a VOPRF mirrors the design of Jarecki *et al*. for password-protecting secret sharing (further developed in more recent work). Other recent and related work includes the *Verifiable Random Functions* developed by Goldberg et al. and the work of Burns et al. for constructing elliptic curve oblivious pseudorandom functions. A VOPRF protocol allows a server with key `x` and a user with input `y` to evaluate `F_x(y)` for some PRF `F`, without the user learning `x` and the server learning `y`. This is the oblivious aspect of the construction. The verifiable aspect allows the user to verify that what is returned by the server is a valid pseudorandom output from the PRF. That is, to prevent the server from returning some input that is potentially not random.

For our design, we view the action of computing `F_x(y)` as an authenticator for `y`. For example, a user can present `y` along with `F_x(y)` and a service with `x` can verify that the

user has received a valid `F_x(y)` in the past. Notice that our interaction is actually symmetric in operation, only users with access to the key `x` can verify. We also introduce two additional properties that we require for our specific use-case:

- Allow blinding to be incorporated into the VOPRF protocol

There are client-side procedures `blind()` and `unblind()` such that `unblind(F_x(blind(y)) = F_x(y)`. This allows us to prevent an adversarial service from linking `(y,F_x(y))` to any invocation of the VOPRF protocol. We call this property 'unlinkability'.

- Verifiable key consistency

We use a batched non-interactive, zero-knowledge 'proof' (NIZK) that allows the service to prove that all clients are served outputs from the VOPRF using the same key `x`. A service that could use unique key pairs for each invocation of the VOPRF protocol would be able to link future redemptions by analysing which key pair was used to compute the VOPRF protocol previously. We instantiate this NIZK using a batched discrete-log equivalence (DLEQ) proof as described by Henry. This is equivalent to verifying that the server output is random.

Below we give the explicit steps and operations required for instantiating the different components of the protocol. We provide a more detailed breakdown of the protocol design here.

Steps

We use an elliptic-curve-based group setting to construct our protocol between a client and a server. The goal of the protocol is for the client to receive tokens that are multiplied by a secret,

server-chosen value. This is what we refer to by 'signing'. In the protocol, we assume that the client performs a successful authentication and seeks to gain a number of signed tokens (in this case 1) that can be redeemed for similar authentications in the future.

In the below, `H_1` and `H_2` are hash functions mapping onto, respectively, the group and `{0,1}^L` where `L` is a security parameter (we also use a third hash `H_3` for the DLEQ proof in step 3). The hash functions are modelled as random oracles (this is cryptographic terminology so that we can assume that the outputs are uniformly distributed). The server also publicly publishes a generator `G` along with a commitment (or 'public key') `Y=xG`, these are also used for the DLEQ proof step.

1. Client generates a random token `t` and a blinding factor `r`.
2. Client calculates `T = H_1(t)` and sends `M = rT` to the server along with a means of authentication (such as a proof-of-work solution).
3. Server validates the solution and computes `Z = xM =rxT` and a DLEQ proof `D`.
4. The server returns `(Z,D)` to the client.
5. Client unblinds `Z` to retrieve `N = (1/r)Z = xT` and stores the pair `(t,N)`.
6. When the client wants to redeem a token it presents `(t,MAC(N,request-binding-data))` where `N` is a shared key for the `MAC` algorithm and `request-binding-data` is made of information observable by the server that is unique(ish) to that particular request.
7. The server uses `T` as a double-spend index and recalculates `N` using `x`. Then it can validate the MAC using the shared key `N`.

8. We know that a matching commitment value is valid because generating it requires access to `x`.

We think of steps 1–5 as the signing phase of the protocol and steps 6–8 as the redemption phase. The `request-binding-data` that we use above is application-specific data that can be used to prevent the request from being hijacked. Think of a redemption occurring over HTTP, for example.

DLEQ proofs

In step (3.) above, we call for a zero-knowledge proof of the equality of a discrete logarithm (our server key) with regard to the returned token `Z` that the client receives. *The protocol that we give below is finished but the DLEQ verification in the browser extension in still under development. In particular, the proof verification is not completely consistent with the specification here —we specify more details below.*

A DLEQ proof allows the client to verify that the same key `x` is being used to 'sign' all tokens that are sent to the server. For a group of prime order `q` with generators `M`, `G`, public key `Y`, and point `Z`:

1. Prover chooses a random value (also known as a 'nonce')

```
k <-- Z/qZ
```

2. Prover commits to the nonce `k` with respect to both generators

```
A = kG, B = kM
```

3. Prover constructs the challenge

```
c = H_3(G,Y,M,Z,A,B)
```

4. Prover calculates response

```
s = k−cx (mod q)
```

5. Prover sends `(c,s)` to the verifier

6. Verifier recalculates commitments

```
A' = sG + cY
B' = sM + cZ
```

7. Verifier hashes

```
c' = H_3(G,Y,M,Z,A',B')
```

and checks that `c' == c`.

If all users share a consistent view of the tuple `(G,Y)` for each key epoch, they can all prove that the tokens that every client has been issued share the same anonymity set with respect to `x`.

Batching proofs

In practice, the issuance protocol operates over sets of tokens rather than just one. A system parameter, `m`, determines how many tokens a user is allowed to request per valid authentication. Consequently, users generate `(t_1,t_2,...,t_m)` and `(r_1,r_2,...,r_m)`; send `(M_1,M_2,...,M_m)` to the server; and receive `(Z_1,Z_2,...,Z_m)` in response.

Generating an independent proof of equality for each point implies excess overhead in both computation and bandwidth consumption. Therefore, we employ a batch proof to show consistent key usage for an entire set of tokens at once. The proof is a parallelized Schnorr protocol for the common-exponent case taken from the work of Henry and adapted for non-interactivity:

Given `(G,Y,q); (M_1,...,M_m), (Z_1,...,Z_m);` where `Z_i = x(M_i)` for `i=1...m`:

1. Prover calculates a seed using:

```
z = H_3(G, Y, M_1, ... , M_m, Z_1, ... ,
Z_m)
```

2. Prover initializes `PRNG(z)` and samples from it to non-interactively generate

```
c_1, … , c_m <-- Z/qZ.
```

3. Prover generates composite group elements `M` and `Z` by computing:

```
M = (c_1*M_1) + (c_2*M_2) + ... +
(c_m*M_m)
Z = (c_1*Z_1) + (c_2*Z_2) + ... +
(c_m*Z_m)
```

4. Prover sends proof

```
(c, s) <-- DLEQ(Z/M == Y/G)
```

5. Verifier recalculates the PRNG seed from protocol state, generates the composite elements, and checks that `c' == c` as in the single-element proof above.

We can see why this works in a reduced case.

For `(M_1,M_2)`, `(Z_1,Z_2)`, and `(c_1,c_2)`:

```
Z_1 = x(M_1)
Z_2 = x(M_2)
(c_1*Z_1) = c_1(x*M_1) = x(c_1*M_1)
(c_2*Z_2) = c_2(x*M_2) = x(c_2*M_2)
(c_1*Z_1) + (c_2*Z_2) = x[(c_1*M_1) +
(c_2*M_2)
```

So the composite points will have the same discrete log relation `x` as the underlying individual points.

[Currently the extension does not support the re-computation of the PRNG state as above and so we require that the server sends this with the batch proof. The batch proofs are still in development and we hope to finish support for complete verification in the near future.]

Security

The security of our protocol rests upon some key security goals:

- The server cannot link redemption requests to any invocation of the signing phase
- The client is not able to learn the secret key of the server during the signing phase
- The server cannot cheat using the DLEQ proof to suggest that the key pair is consistent, when it is not.

Cryptographically our protocol satisfies these three requirements using cryptographic blinding, the hardness of computing the discrete log problem for the elliptic curve that we use and the soundness of the NIZK DLEQ proofs that Henry showed, respectively.

There are additional security concerns that occur out-of-band with respect to a cryptographic proof but are nonetheless harmful. For example, one concern is that it is possible to stockpile tokens over a period of time and the redistribute these (since tokens are not bound to the user to prevent deanonymisation). We discuss these possible attacks in greater detail in the main protocol specification.

Overheads

In preliminary tests on consumer hardware, our extension takes ~1.1 seconds to generate blinded tokens to be signed by the server and ~1.9 seconds to parse the signed tokens and verify the DLEQ proof. Creating a pass that can be used to redeem signed tokens takes <40ms. In terms of request sizes, Privacy Pass adds ~2kb of data to client requests for tokens to be signed and ~0.4kb for requests to redeem a pass. The server signing response includes ~17kb additional data.

Therefore, Privacy Pass is unlikely to add any disruption to current client browsing and also can be used to minimise direct human interaction that would have previously taken much longer.

Support for Privacy Pass

The internet security and performance company Cloudflare is the first partner to implement support for Privacy Pass. Using Privacy Pass with Cloudflare websites allows users to acquire signed 30 tokens for each Cloudflare challenge CAPTCHA that they solve. Once tokens are acquired, users are able to bypass future Cloudflare CAPTCHAs using the redemption phase of the protocol above.

Cloudflare is a prominent presence on the internet and their support is likely to make Cloudflare-protected websites much more accessible for all users that are affected by having to solve these CAPTCHAs. Indeed, Cloudflare challenges are disproportionately likely to target clients who use shared IPs when browsing. Our solution means that these users will be able to drastically reduce the number of CAPTCHAs they have to solve during navigation.

We hope that we can acquire more partners to support Privacy Pass in the future. For example, we think another potential use case for the extension could be 'anonymous' authentication for content-creating spaces. This could include logging into a service using a pass rather than an email and password. Furthermore, in a wiki-style environment, it could be possible to allow users to authenticate with Privacy Pass and thus allow anonymous editing for the wiki. If you are interested in implementing server-side support for Privacy Pass then get in contact with the Privacy Pass team.

Download

Privacy Pass is available as a browser extension for both Chrome and Firefox.

Code and contributing

On GitHub, we have open-sourced a compatible server implementation that is written in Go so that anyone can set up a server that supports Privacy Pass. We have also open-sourced the code for Privacy Pass itself and we encourage contributions from the wider community if you think there is something that

we can improve. All code is open-sourced under the BSD-3 license.

If you would like to know more about the design decisions and the work that was done to create Privacy Pass then see our website.

Found an issue?

There may be issues that you find when using Privacy Pass as it is effectively still in beta. If you have spotted anything that you think is wrong then please get in touch using any of the links above.

Acknowledgements

The creation of Privacy Pass has been a joint effort by the team made up of George Tankersley, Ian Goldberg, Nick Sullivan, Filippo Valsorda and myself.

We would also like to thank Eric Tsai for creating the logo and extension design, Dan Boneh for helping us develop key parts of the protocol, as well as Peter Wu and Blake Loring for their helpful code reviews. We would also like to acknowledge Sharon Goldberg, Christopher Wood, Peter Eckersley, Brian Warner, Zaki Manian, Tony Arcieri, Prateek Mittal, Zhuotao Liu, Isis Lovecruft, Henry de Valence, Mike Perry, Trevor Perrin, Zi Lin, Justin Paine, Marek Majkowski, Eoin Brady, Aaran McGuire, and many others who were involved in one way or another and whose efforts are appreciated.