Idiot-Proofing Authentication

Jacob Hurwitz *jhurwitz@mit.edu*
Will Oursler *woursler@mit.edu*
Michael Sanders *mssand@mit.edu*

December 13, 2013

1 Abstract

We develop a mechanism by which a server can guarantee that client passwords contain a certain amount of (actually random) entropy while not actually knowing these passwords. We define and implement a protocol between a client and server used during password generation. Importantly, while we greatly modify the process of generating a password, the actual day-to-day use of this password is mostly unmodified, allowing for the use of additional password protection schemes on top of our system, like SRP. We describe a specific protocol which can be used with a minimally modified version of SRP to provide strong guarantees on the security of user passwords.

2 Motivation

"The Quest to Replace Passwords" makes a compelling argument that passwords are the best authentication scheme currently available[1]. Unfortunately, the security of passwords is often undermined by users who re-use the same password on dozens of services. A website that wishes to be highly secure can force users to pick a password that is independently secure (e.g., contains a mix of letters, numbers, and symbols), but that website has no way of guaranteeing that the user is not picking a password that they already use on other sites. Requiring password entropy is possibly the only way for a server to prevent password re-use.

Another motivation is that both the client and the server may be worried about adversaries trying to falsely authenticate as the user. In the worst case, an attacker might have access to the authentication database (e.g., hashes, salts) stored on the server. The Stanford Remote Password Protocol (SRP) addresses this threat model; even an attacker who can eavesdrop on all SRP communications and knows fully the information held by the server cannot impersonate the client[5].

3 Threat Model

We will assume that we need not be overly concerned with either the security/integrity of client-server communication or the integrity of the authentication service, though in practice these are both vital to secure operation. We are instead more concerned with the limited degree of trust each of the server and client can afford each other during the password generation process.

3.1 Server Threat Model

We imagine a server faced with a client who, for their own unspecified reasons, wishes to pick the least secure password available to them—i.e., a short password, or some commonly-used words. We do not assume there is necessarily malicious intent on the part of the client, although there could be. Perhaps, given the option, the client would like to fully determine their password to enable password re-use or memorability.

The server itself, however, wants clients to use only secure passwords (those with a large amount of entropy) to ensure that the use of their service is not vulnerable to a breach due to poor password choices on the part of their clients. They therefore do not want the security and entropy of passwords to be subject to the choices made by clients.

We also assume that, while a server can make its own authentication process as secure as it desires, the server has no control over other services that the client uses, so the server must assume that every password the client has used on other services has already been compromised.

3.2 Client Threat Model

We adopt the threat model of SRP, with a few modifications.

We assume the client is, from their own perspective, interested in ensuring that the server cannot have enough information to impersonate the client to someone else. We imagine that the client is concerned about the potential for breaches in the server, and so do not want to be required to provide it with information sufficient to reconstruct their new password or a password-equivalent (such as a hash, if that hash alone is sufficient to authenticate to the server). There are two components to implementing this wish: one, that the password is sufficiently random, and two, that the server does not have the password itself.

It is reasonable to ask why we are concerned about breaches of the verification service when it could be argued that such breaches indicate that the server is already compromised beyond help. We have several related reasons why we take this approach.

1. Most websites and web frameworks - particularly the security-conscious websites we're targeting with our proposal - want authentication systems that are extremely compact and well-audited for potential security vulnerabilities. A security-conscious website would likely not want to modify its existing system to accommodate a new, experimental protocol. To provide some assurances, we will propose making a new service that websites run called the "verification service" that verifies that the password meets the entropy requirements. This service can be carefully sandboxed so that even if it is compromised, it cannot seriously affect the rest of the system. Our goal then becomes ensuring that should the verification service be compromised, client passwords are not revealed to third parties.

2. Assuming the service is compromised in such a manner neatly subsumes the set of possible attacks we might concern ourselves with if the server has poor RNG. We can implicitly assess the types of attacks that might be possible, since the sorts of attacks possible by exploiting poor RNG can always be simulated by imagining that an adversary runs the correct algorithms on the compromised service, but carefully picks or logs the numbers being generated.

4 Discrete Log Commitments

The intractability of the discrete log problem underpins the security of both SRP and our own protocol [1]. The problem can be stated as follows: given elements g, y of a group G, find some integer x such that $y = g^x$. This problem is believed to be generally intractable, but given g and x, it is obviously not difficult to compute $g^x = y$. Therefore, the discrete log also yields a rather convenient one-way function. As RSA depends on the hardness of factoring, our scheme depends on the hardness of the discrete log problem.

A commitment to a particular value is, abstractly speaking, the publication of information that can later prove that the publisher knew a particular value at the time of the commitment. We might, for example, wish to bid in silent auction, but not reveal how much money we are willing to pay for fear that we will be one-upped by our competitors. During bidding, instead of revealing our bid we publish a commitment to our bid. After bidding has closed, we can reveal our bid ("unwrapping our commitment"), and interested parties can verify that the bid we published does in fact correspond to the bid we have revealed. In this way, we can make a secret bidding system with no trusted entity.

[1] This is no coincidence. It is precisely because both our schemes are based on discrete logs that we are able to dovetail them.

A simple commitment scheme involving the discrete log problem is to let the commitment of a value x be $C(x) = g^x$. However, this scheme is undesirable because it's deterministic; someone can learn whether two commitments are for equal values by comparing the commitments. Semantic security can be introduced by generating a (secret) random value r for every commitment, and then letting a commitment be $C_r(x) = g^x y^r$, where $g = y^z$. To unwrap a commitment, we publish both x and r[2].

It's important to consider which parties have knowledge of z. In a typical zero-knowledge proof system, there is one "prover" and one "verifier," and only the prover makes commitments. In our system, both the client and server will need to make commitments, so neither one of them can have complete knowledge of z. Having z would violate the computational binding property because this allows for a prover to find two values, x and $x - k$, for the same commitment: $g^x y^r = g^k g^{x-k} y^r = g^{x-k} y^{r+zk}$. We created a system for the client and server to cooperate in a choice of z so that neither party knows its value, but both agree upon it.

To use the terminology of Damgård et. al, our system has the computational binding and unconditional hiding properties, with respect to the client as the prover[3]. Unconditional hiding is very important to us because the server will have the commitment to the client's password and, even with arbitrary computing resources, we want to ensure that the server never learns the client's password. We can accept computational binding (in lieu of unconditional binding) because we assume the client will not have enough computing resources to change its mind about a commitment during the protocol exchange, and finding an equivalent password after-the-fact will not advantage the client.

Importantly, these commitments are partially homomorphic; $C_{r_1}(x_1) \cdot C_{r_2}(x_2) = C_{r_1+r_2}(x_1 + x_2)$.

5 Protocol

We describe a protocol by which a client can prove to a server that its password contains entropy provided by the server, without ever revealing the password to the server. The "server" in this protocol could be any web server. We imagine that a secure web server would have privilege-separated services. Because this protocol does not use or need authentication data, we assume that a privilege-separated web server would establish a completely new privilege-separated service (called the "verification service" or alternatively the "password generation service") to conduct this protocol; afterwards it would communicate with the authentication service to confer approval of the client's password.

The protocol, as depicted in Figure 1, is:

1. The client initiates the protocol by sending a username to the server.
2. The server generates a prime N, a prime q such that $q | N - 1$, and an element y such that y has order q in $\mathbb{Z}_N^*$. The domain of commitments will be $\mathbb{Z}_q$ and the range will be $\mathbb{Z}_N^*$, and all arithmetic is conducted in the appropriate group. The server also generates a partial private key, z_S. The server sends N, q, y, and y^{z_S} to the client.
3. The client generates a partial private key z_C and sends g^{z_C} to the server. At this point, both the client and server compute $g = y^{z_S + z_C} = y^{z_S} y^{z_C}$ (We can see that g also has order q.).
4. The server generates a random value A and its commitment $C_{r_A}(A)$ and sends the commitment to the client.
5. The client generates a random value B and its commitment $C_{r_B}(B)$ and sends the commitment to the client.
6. The server reveals its commitment to the client by sending A and r_A.
7. The client verifies that the commitment was revealed correctly.
8. The client computes $C = A + B$ and generates commitment $C_{r_C}(C)$ using new randomness r_C. The client sends this commitment to the server.

[2]We're actually leaving out a lot of the mathematical details here; see Cramer et. al. for a full explanation[2].

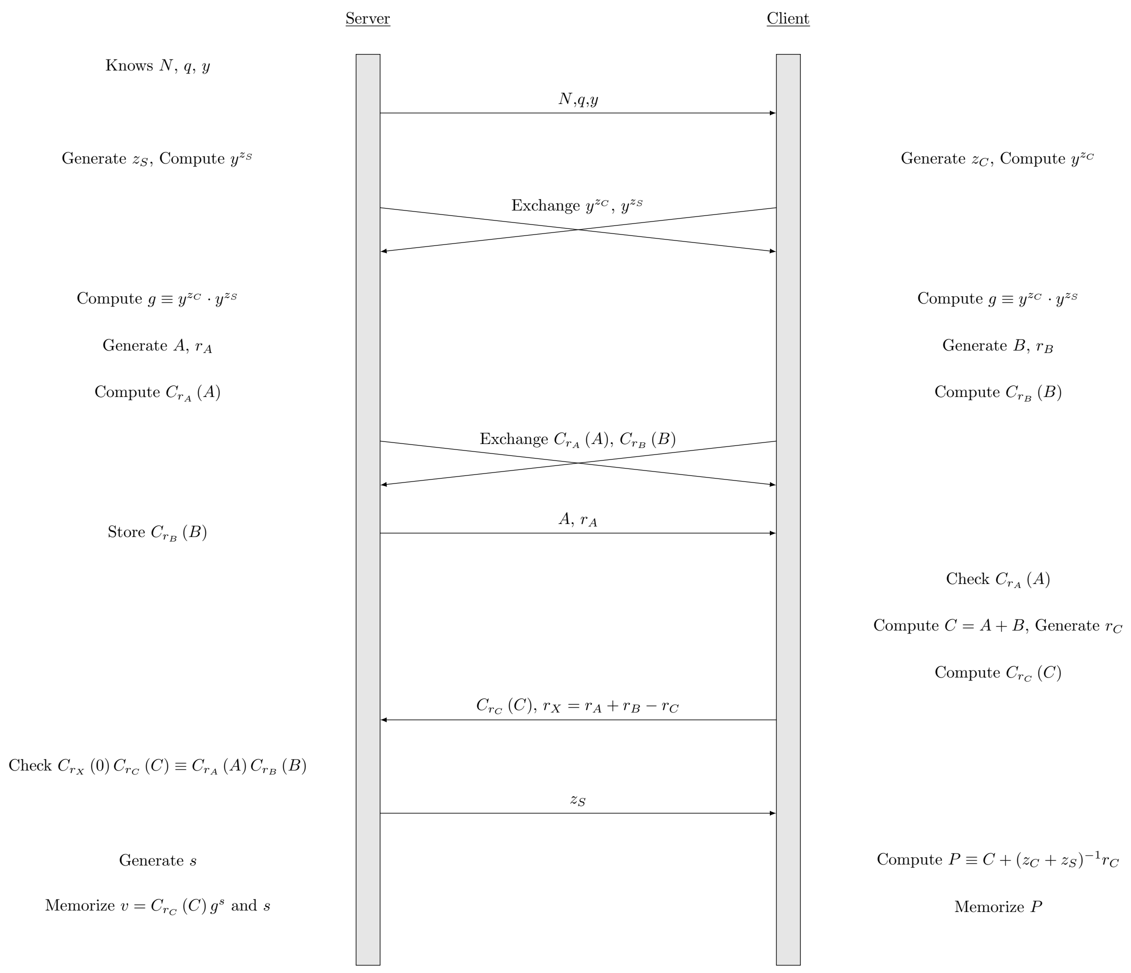
Server
Client
Knows N, q, y
N,q,y
Generate z_S, Compute y^{z_S}
Generate z_C, Compute y^{z_C}
Exchange y^{z_C}, y^{z_S}
Compute $g \equiv y^{z_C} \cdot y^{z_S}$
Compute $g \equiv y^{z_C} \cdot y^{z_S}$
Generate A, r_A
Generate B, r_B
Compute $C_{r_A}(A)$
Compute $C_{r_B}(B)$
Exchange $C_{r_A}(A)$, $C_{r_B}(B)$
Store $C_{r_B}(B)$
A, r_A
Check $C_{r_A}(A)$
Compute $C = A + B$, Generate r_C
Compute $C_{r_C}(C)$
$C_{r_C}(C)$, $r_X = r_A + r_B - r_C$
Check $C_{r_X}(0)\,C_{r_C}(C) \equiv C_{r_A}(A)\,C_{r_B}(B)$
z_S
Generate s
Compute $P \equiv C + (z_C + z_S)^{-1} r_C$
Memorize $v = C_{r_C}(C)\,g^s$ and s
Memorize P

Figure 1: Our Protocol

9. The client also computes $r_X = r_A + r_B - r_C$. Revealing r_X does not reveal either of r_B or r_C, so we send r_X to the server.

10. The server checks that $C_{r_X}(0)\,C_{r_C}(C) = C_{r_A}(A)\,C_{r_B}(B)$. If this checks out[3], then the server knows that the client correctly set $C = A + B$.

11. The server sends z_S to the client. This lets the client reconstruct the full private key $z = z_C + z_S$, but that's okay because the server and client are done swapping commitments, and the server is not hiding any values from the client.

12. The client computes $P = C + (z_C + z_S)^{-1} r_C$ and stores P (e.g., in a password manager) as its password.

13. The server randomly generates a salt s and computes $v = C_{r_C}(C)g^s$. The server stores v as the SRP verifier for this user. g, unique to each user, is also stored.

[3]Because C was calculated as the sum of A and B (modulo N) C has at least as much entropy as each of A and B. Thus, C will be the basis for the client's password. If all we wanted to do was generate a password then we could stop here and use C, but we need a few more steps in our protocol in order to integrate with SRP.

In summary, this protocol lets the client and server mutually agree on a password, prove to one another that it contains as much entropy as each one contributed, and lets the server compute an SRP verifier for this password—all without the server ever receiving the password or any password-equivalent data!

6 Integration with SRP

At the conclusion of our protocol, each party knows exactly the information it needs to run SRP.

We had to make small modifications to SRP-6a to integrate it with our system[6]. These modifications have some security implications, which we consider here.

First, where as standard SRP uses the same g for all users, we now have a unique g for each user. In several ways, this represents a security boon.

Standard SRP computes a private key as $x = H(s, P)$, the hash of the user's salt and password. In our system, we compute this private key as $x = s + P$, using the sum of the values instead of a hash. We believe that this is still secure for two reasons. First, x only ever appears as an exponent, so by the hardness of the discrete log problem, an attacker who compromises the database (including verifiers) still cannot compute x. Additionally, our system is secure against rainbow table and association attacks for x because g is different for each user, and the server does not know conversion factors.

Additionally, the SRP specification requires N to be a "safe prime" (meaning $N = 2q + 1$ with q prime) and for g to be a generator modulo N. The authors of SRP clarify: "For SRP, we wish to maximize the difficulty of calculating discrete logarithms in $GF(n)$. For this reason, n must be a non-smooth prime, which means that $n - 1$ must not consist entirely of small factors [...] Since the probability of generating a smooth prime at random is quite small [11], n can, in practice, be safely generated by selecting a random, large prime. Nevertheless, for maximal security, the author recommends that n be a safe prime"[5].

We pick N such that q divides $N - 1$, where N has 2048 bits and q has 160 bits. (The more bits q has, the more non-smooth N is.) Recent computational results have shown that, as of 2012, the discrete log problem can be solved in about 24 hours if q has 80 bits; but extrapolating from the data presented in the paper, it would currently take about 10^6 years to solve the discrete log problem modulo N when q has 160 bits[4]. Thus, our choice of non-safe prime N still satisfies the author's requirement that N be non-smooth in order to maximize the difficulty of calculating discrete logarithms modulo N.

As for g not being a generator, the rationale for requiring g to be a generator was: "Additionally, g must be a primitive root of GF(n) in order to make all values of B equiprobable for any v. If this requirement is not met, a partition attack again becomes possible"[5]. The partition attack described occurs when an adversary poses as a user and makes many, many incorrect password attempts in order to gain as much data about the verifier as possible. Although we make a partition attack slightly easier by choosing g to have order q, we can easily counter by rate-limiting incorrect password attempts. This would make a partition attack essentially impossible since an adversary would not be able to gain enough data to mount such an attack.

7 Code

Code for our project can be found on Github at `https://github.com/Zomega/lab6858/tree/commitments`, or by cloning the repository from `git@github.com:Zomega/lab6858.git` and checking out the `commitments` branch. This paper is included under `/documentation/`.

We wanted to write our client-side cryptography in Python, so the website contains Javascript that talks to a local Python daemon over a websocket. Thus, the client must have this daemon installed and running in order to register for or log into the website. (In the future, this problem could be avoided by rewriting these programs in pure Javascript.) We highly recommend that the user also install and run a password manager of his/her choosing in order to securely store the randomly-generated password. Most password managers will automatically prompt to save the password upon submission of the log-in page, and for those that don't, we conveniently copy the password to your clipboard.

We also wrote a very barebones web server, nicknamed "ZooBank," to demonstrate our password generation and modified SRP protocols. In order to allow the reader to focus on our unique protocol, we made the web server as simple as possible, at the expense of introducing problems that would need to be fixed before implementing our protocol on production code. (We kept a list of such bugs in `known_bugs.txt`.) The web server is written in Flask, and in order to aid our prototyping process, all data is stored in-memory rather than in a database.

We also wanted to demonstrate that it was possible to privilege-separate the authentication, password generation, and static/dynamic serving services. Rather than write our own privilege separation code (as in Zoobar), we merely implemented each service as its own Flask app running on a different port. As such, running the web server involves starting three separate Python programs, as detailed in `README.md`. In fact, our extreme privilege separation shows that it's possible for the password generation service to be run by a trusted third party that is completely separated from (and run by a different party than) the main web server.

The code is fairly well self-documented, though some of the variable names differ from what was presented in this paper. Our password generation protocol is implemented in `client/client_generate.py` (on the client-side) and `server/bank_gen.py` plus `server/pwgen_server.py` (on the server-side), and our modified SRP protocol is implemented in `client/client_login.py` (on the client-side) and `server/bank_auth.py` plus `lib/srp_patches.py` (on the server-side).

References

[1] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, Frank Stajano, *The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes.* Proc. IEEE Symp. on Security and Privacy 2012.

`http://research.microsoft.com/pubs/161585/QuestToReplacePasswords.pdf`

[2] Ronald Cramer and Ivan Damgård, *Zero-Knowledge Proofs for Finite Field Arithmetic or: Can Zero-Knowledge be for Free?.* Advances in Cryptology: 18th Annual International Cryptology Conference 1998.

`http://www.brics.dk/RS/97/27/BRICS-RS-97-27.pdf`

[3] Ivan Damgård and Jesper Buus Nielsen, *Commitment Schemes and Zero-Knowledge Protocols.* Cryptographic Protocol Theory 2011.

`https://services.brics.dk/java/courseadmin/CPT/documents/getDocument/ComZK08.pdf?d=86909`

[4] Ian Goldberg and Ryan Henry, *Solving Discrete Logarithms in Smooth-Order Groups with CUDA.* SHARCS 2012.

`http://cacr.uwaterloo.ca/techreports/2012/cacr2012-02.pdf`

[5] Tom Wu, *The Secure Remote Password Protocol.* Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium.

`http://srp.stanford.edu/ndss.html`

[6] Tom Wu, *SRP-6: Improvements and Refinements to the Secure Remote Password Protocl.* 2002.

`http://srp.stanford.edu/srp6.ps`