PAuth: A Peer-to-peer Authentication Protocol

Zijing Gao, Thomas Lu, Anand Srinivasan

March 20, 2015

Abstract

PAuth is an authentication protocol in which users' identities are verified by their peers, who subsequently report the results of the verification to a central server or system. Our aim in designing PAuth was to defend against both DoS attacks based on the computational cost of password verification and brute-force attacks on hashed passwords: servers that hash passwords using a computationally expensive hash function are vulnerable to the first attack, while servers that use a cheap one are vulnerable to the second. PAuth completely obviates the need for servers to store hashed passwords, but trades the decrease in computational responsibility for an increase in communicative responsibility.

1 Introduction

Username-password is one of the most common ways to authenticate users to a central service. Typically, on the server side, this involves storing storing pairs of the form $(U, H(P))$, where U is the username, P is the password, and H is a one-way, collision-resistant hash function. When a user submits an authentication request with username U and password P', the server then checks if $H(P') = H(P)$. The hash function H prevents an attacker with access to the username-password table from gaining access to users' accounts. Another common practice is to also include a pseudorandom salt S with each pair and store triplets of the form $(U, S, H(S|P))$, where $|$ denotes concatenation, and checking whether $H(S|P') = H(S|P)$ when a user submits an authentication request (U, P'); this provides additional protection against rainbow table attacks.

In most systems employing username-password authentication, a central server (or centrally-controlled distributed system) performs all hashes for password checking. Thus there is a significant tradeoff to be made in the choice of hash function. A computationally cheap hash function allows attackers to perform brute-force attacks more quickly; according to [1], users choose passwords with 40.54 bits of security on average. If the SHA-256 hash function were used to hash passwords, then using an AntMiner S5 ASIC machine, which achieves 3.12×10^9 hashes per second [2], a well-mounted brute force attack can theoretically crack the average password in $2^{40.54}/3.12 \times 10^9 = 512$ seconds on expectation (assuming good knowledge about password distributions). However, a computationally expensive hash function opens up an opportunity for a denial-of-service attack: by sending many authentication requests, perhaps using a botnet, an attacker can overwhelm the computational capability of the system by forcing it to perform a large number of hashes. Such attacks have been mounted against wireless sensor networks [3], and although it would be difficult to perform a similar attack on a larger system, it is not unimaginable.

PAuth is an authentication protocol that obviates the need for the central system to perform computational verification of a password. Instead, a user's identity is verified by his/her peers via zero-knowledge proofs. While PAuth lightens the computational load on a system, it increases the communication load; however, there is evidence that network communication is becoming less costly in computer systems [4], and we are hopeful that our protocol can evolve into a practical authentication scheme in the future.

Section 2 presents the protocol. Section 3 outlines some of the design decisions that we made, and section 4 describe attacks that we considered and possible enhancements we can make to mitigate their risks. Section 5 outlines an estimation of good values for two key parameters in an actual large-scale deployment of PAuth. Section 6 describes the current state of implementation, and section 7 concludes.

2 Protocol

When a user registers an identity with a server running PAuth, the user generates a public/private key pair (PK, SK) and sends the public key PK to the server along with any other necessary identification information (username, etc.). The authentication protocol then proceeds as follows:

1. When the user wishes to authenticate, he/she sends PK to the central server.

2. The server randomly selects k peers that are currently connected to the server, and sends the networks addresses of these peers $(IP_1, IP_2, \ldots, IP_k)$ back to the user.

3. The authenticating user contacts each of these peers and sends PK to them to begin a zero-knowledge proof of possesion of SK.

4. Each peer generates a random string N_{peer}, encrypts it with the public key to get $E_{PK}(N_{\text{peer}})$, and sends the result back to the authenticating user.

5. The user replies to each peer with the decryption $D_{SK}(E_{PK}(N_{\text{peer}}))$.

6. Each peer checks if the user's reply is equal to the original nonce, and sends either (YES, PK) or (NO, PK) to the server accordingly.

7. If the server receives at least k' replies of the form (YES, PK) within a timeout period t, then the authentication attempt succeeds. Otherwise, the authentication attempt fails.

A graphical representation of this protocol is provided in Figure 1. The parameters k and k' are adjustable to provide the best combination of security and efficiency. We discuss possible values of these parameters and their implications in section 5.

Note that the protocol requires at least k users to be connected to authenticate any additional users. The first k users can be connected to the system using other methods, or we can seed the user pool with a set of "initial peers" that are deployed alongside the central server or system. This solves the problem of initialization.

This protocol provides no security guarantees about any individual message. Instead we rely on security in numbers, similar to the Bitcoin system. Just as the security of Bitcoin depends on attackers being unable to mount a majority of computational power [5], the security of our protocol depends on attackers being unable to maintain a significant fraction of connections to the central server. This issue is discussed further in section 5.

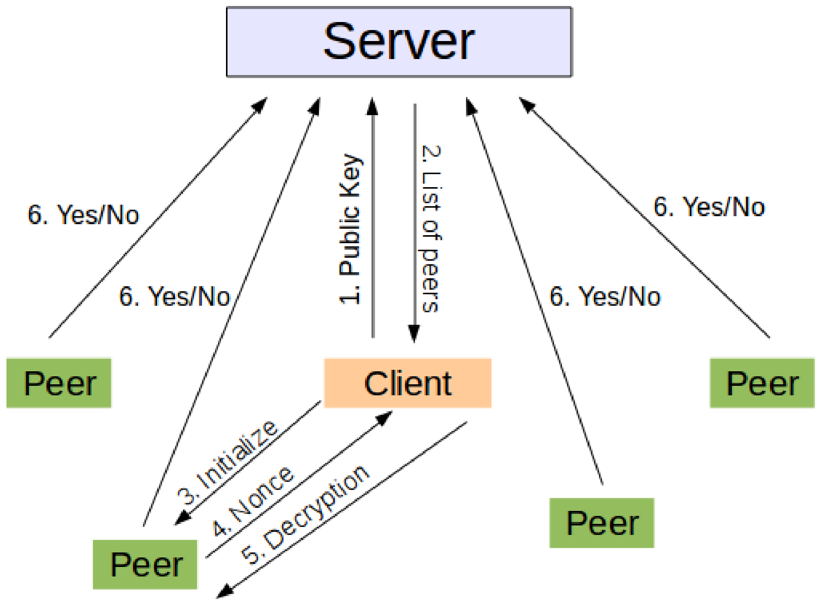

Figure 1: This is a graphical representation of the protocol described in section 2. For simplicity, only communication with one peer is depicted in the figure, and only four peers are displayed. In an actual implementation, a client would attempt to communicate with all peers named by the server, and the server would name more than four peers.

3 Design Decisions

In the course of designing this protocol, we considered a number of possible options for each part of the protocol. We present some of the more significant decisions that we made and the reasons for these decisions.

3.1 Peer Contacting

In an earlier version of the protocol, communication with the peers of an authenticating user was initiated by the server, and communication between authenticating user and peers was initiated by the peers. This presented an opportunity for a denial-of-service attack against users: by submitting a large number of bogus authentication requests on behalf of a user to the server (this is easy to do, as this simply involves sending the public key of that user), an attacker can cause kr packets to be sent to the user by only sending r packets him/herself, easily flooding the user's network connection if k is modestly large. Rate-limiting by machine could mitigate the influence of an attacker with a single machine, but would not stop one in control of a botnet. Rate-limiting by user could protect a user from network flooding by a distributed attacker, but also introduces a method for attackers to deny opportunities for authentication to a user.

Instead, by shifting the responsibility of peer contacting to the authenticating user, we remove the possibility of a flooding attack employing PAuth. Users can simply ignore peer lists that they did not expect to receive from the server, and so the PAuth protocol makes it no easier for an attacker to flood a user's network.

3.2 Authentication Initialization

We considered a number of ways for users to contact the server to indicate intent to authenticate. We initially wanted to find a scheme that would prevent attackers from submitting bogus authentication requests on behalf of a user. The alternative that we considered most seriously was the addition of another secret "identity key" to the protocol that users would submit to the server to indicate intent to authenticate. However, to minimize computational load on the server (which is the ultimate intent of this protocol), the identity key would have to be sent in cleartext, and could thus be stolen easily. We therefore decided to indicate intent to authenticate using publicly available information, and prevent or reject bogus requests using different methods (specifically, the one described above).

4 Possible Attacks and Mitigations

In this section we discuss possible attacks against this protocol and steps that we can take to mitigate the risks of these attacks. Not all of these measures have been implemented, but it would not be difficult to do so.

4.1 Computational Overload of Peers

In the protocol described, a peer that is connected to the server and receives a request to begin a zero-knowledge proof has no way of knowing whether the request is legitimate. By sending a peer many requests of this form, we can force the peer to generate and encrypt a large number of random nonces, possibly overwhelming the computational capabilities of this peer. While it is unlikely that an attacker will be able to overwhelm the entire network of peers in this way (as the resources required to mount such an attack grows linearly with the size of the network), it is still undesirable for attackers to be able to bring down the CPUs of chosen peers.

To mitigate the risk of this attack, we can have the server contact peers with the PK of the authenticating user when they are chosen as one of the k verifying peers. Peers can then ignore requests to begin zero-knowledge proofs that they were not expecting. Because the set of verifying peers is chosen randomly by the server for each authentication attempt, it is thus unlikely that any one peer would be overloaded.

4.2 Communication Overload of Server

While our protocol is designed to defend against a CPU-overloading DoS attack on the server, it does little by itself to defend against a DoS attack that overloads the server's communication capabilities. Because this protocol multiplies by a factor of k the number of packets that a server must process for each authentication request, it becomes much easier for an attacker in control of many user identies to mount such an attack. Unfortunately, we have thus far been unable to devise novel methods for defending against such attacks other than those which already exist (traffic filtering, second authentication steps for suspicious access patterns, etc.).

4.3 Malicious YES/NO Responses

In the last step of our protocol, when peers send $(\text{YES/NO}, PK)$ back to the server for a client, there is no mechanism to stop the peer from sending an arbitrary value, regardless of the outcome of the zero-knowledge proof. Malicious peers can either send "YES" to enable unauthorized access to an account or "NO" to deny service to a user.

There are two ways that we can defend against this type of attack. The first is by selecting a sufficiently large k and a suitable value of k'; this will be discussed in greater depth in the following section. The second is by blacklisting peers who return authentication decisions that contradict the server's decision. The second method, however, can be catastrophic if an attacker is able to, at any point, mount a majority of connections to the central server, as the attacker can subsequently cause a large number of "good" peers to be blacklisted and therefore increase the proportion of malicious peers. A refinement could be to blacklist all peers after every authentication for a length of time proportional to the number of fellow peers on the same authentication who disagreed with them. This would, at the very least, slow the progress of an attacker with a modest majority of connections, possibly enough for human intervention.

5 Selecting Values of k and k'

Selecting suitable values of k and k' is crucial to any actual deployment of PAuth on a real system. However, because PAuth is designed to be used as part of a large system with many millions of users connected to it at any given moment, it is difficult to conduct actual field tests to optimize the values of k and k': we would have to either deploy PAuth on or create a service that attracts such a volume of users and a significant number of attackers. We thus use data on an existing system and current knowledge about botnets to derive estimates for good values of k and k'.

We will use Facebook as our example system system in this section to derive our estimates of reasonable values of k and k'. Facebook has 936 million daily active users [6]. We estimate conservatively that at any given point in time, about 20% of them, or around 180 million, have active connections with Facebook. Given that many users are connected almost 24/7 via their mobile phones, we believe that this is a conservative estimate. According to [7], the BredoLab botnet is the largest botnet ever created, with an estimated 30 million bots. Assuming that all of these bots were constantly connected to Facebook, they would comprise 1/7 of the total connections to Facebook (we assume that the previous 180 million users are not colluding with any botnets).

An attacker can deny service to a user by controlling at least $k - k' + 1$ of a set of k selected peers, or grant unauthorized access to a user's account by controlling at least k' of k selected peers. We can approximate the number of malicious peers in a set of k selected peers as a binomial distribution with $p = 1/7$. Let U denote the event that an attacker gains unauthorized access, and D denote the event that an attacker is able to deny service to a user. In selecting our values of k and k', we wish to achieve the following goals:

1. We wish to make $P(U)$ as small as practically possible without compromising the ability for users to log in at all, or making k unreasonably large. This is the most important goal, as allowing attackers to access users' accounts would be catastrophic.

2. We wish to make $P(D)$ reasonably small, but this is of lower priority, as authentication can be retried with a different set of randomly selected peers (and it is possible to code a retry loop into the client-side software so that the process is invisible to users).

3. We wish to keep k at a reasonable number (less than 20) to minimize communication load on the server. However, if this is unachievable while providing low probabilities $P(U)$ and $P(D)$, we can increase k.

k	k'	$P(U)$	$P(D)$
10	7	9.77×10^{-5}	4.27×10^{-2}
10	9	2.16×10^{-7}	0.429
15	10	5.31×10^{-6}	1.33×10^{-2}
15	13	8.15×10^{-10}	0.365
18	12	5.75×10^{-7}	9.07×10^{-3}
18	14	2.55×10^{-9}	0.103
19	15	4.60×10^{-10}	0.124

Table 1: Some of the pairs (k, k') that we considered and the associated probabilities of unauthorized access ($P(U)$) and denial-of-service ($P(D)$). We see that $k = 19, k' = 15$ provides good probabilities while keeping k at a reasonable value.

Table 1 shows some of the values (k, k') that we considered and the associated probabilities $P(U)$ and $P(D)$ that we computed. We see that $k = 19, k' = 15$ yields the probabilities $P(U) = 4.60 \times 10^{-10}$ and $P(D) = 0.124$. We believe that this is an acceptable value of k and that these probabilities are acceptably low. An attacker attempting to gain unauthorized access would have to make over 2 billion attempts on expectation to actually authenticate successfully, and a legitimate user would be able to authenticate successfully in one try 88% of the time. Furthermore, note that these probabilities were calculated assuming the bare-bones protocol described in section 2 is used with no modifications. If instead we implement the modification described in 4.3 as well, these probabilities will be even lower as we take down increasingly many attacker-controlled peers.

6 Implementation

Our cryptographic implementation of our distributed authentication system was built on top of the popular Stanford Javascript Cryptography Library (`https://crypto.stanford.edu/sjcl/`). The SJCL is a NSF-funded, time-tested library in common use in the field. Our software used the SJCL default-provided RSA implementation as its signature scheme.

As we were using the SJCL, we implemented both our client and server using node.js, a fast and lightweight Javascript platform, with a simple MongoDB database keeping track of the server-maintained peer list. Because node.js is a high-visibility open source platform in production use at many large companies, we expect that it should be as secure as any other easy choice for an initial implementation. Our current server implementation would already easily be deployable to cloud web-hosting platforms such as Heroku, and is fully modular as a node.js package, so can be integrated into other projects rather easily.

We were unable to deploy our implementation on a massive scale on different networks, so we collected no scalability data, but the implementation functioned as expected. As we chose to implement quite a small proof of concept, both the client and server combined took about 1000 lines of written code, with many more files that were autogenerated. The code was used to define a client and server such that:

- the client had a keypair and could:
  - request a peer list through login
  - open a WebSocket to listen for successful auth (WS)
  - join a peer list
  - authenticate with another peer
    - receive challenges and respond
    - generate challenges for peers upon request for authentication
  - forward authentication messages to the server upon verification
- the server had an account system and a peerlist and could:
  - give a random subset of the peer list to a client upon login request (GET)
  - receive authentication certificates (POST)
  - give an authentication token to a successful login (WS)

We chose $(k, k') = (19, 15)$, as determined in the previous section.

7 Conclusion

We have presented PAuth, an authentication protocol that utilizes the computational power of a system's users to authenticate other users; our goal was to obviate the need for these systems to perform computationally expensive hash functions on submitted passwords when authenticating users. PAuth achieves this by greatly increasing the communication costs incurred by servers when authenticating users, but we hope that the protocol can be refined and that network communication will become cheaper in the future. We have presented vulnerabilities that our protocol introduces and have given methods that we devised to counter them. PAuth is still in a very early proof-of-concept stage, and we hope that it will one day evolve into a deployable mechanism.

References

[1] Florencio D, Herley C. 2007. A Large-Scale Study of Web Password Habits. Proceedings of the World Wide Web Conference Committee.

[2] Mining hardware comparison [Internet]. [updated 2015 Apr 1]. The Bitcoin Wiki; [cited 2015 May 12]. Available from `https://en.bitcoin.it/wiki/Mining_hardware_comparison`

[3] Ning P, Liu A. 2008. Mitigating DoS Attacks against Broadcast Authentication in Wireless Sensor Networks. ACM Journals 4(20):1-31.

[4] Nightingale EB, Elson J, Fan J, Hoffman O, Howell J, Suzue Y. 2012. Flat Datacenter Storage. Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12); 2012 Oct 7-10; Hollywood.

[5] Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. Avaiable from `https://bitcoin.org/bitcoin.pdf`

[6] Facebook Reports First Quarter 2015 Results [Internet]. [updated 2015 Apr 22]. Facebook; [cited 2015 May 12]. Available from `http://investor.fb.com/releasedetail.cfm?ReleaseID=908022`

[7] Botnet [Internet]. [updated 2015 Apr 24]. Wikipedia; [cited 2015 May 12]. Available from `http://en.wikipedia.org/wiki/Botnet`