Zero-Knowledge Proof PAM SSH

Massachusetts Institute of Technology

6.857: Computer and Network Security

Noah Flowers, Sylvie Lee, Christine Vonder Haar

{nflowers, sylviel, vondy} @ mit.edu

May 15, 2019

Abstract

This paper explores an exciting alternative to the typical password-centered authentication that is so often used today. Throughout the process of building the system, we investigate the concepts of Zero-Knowledge Proofs, Elliptical Curve Cryptography, Interactive and Non-interactive Protocols, and Pluggable Authentication Modules (PAMs). We explain and implement the Schnorr Zero-Knowledge Proof Non-interactive Identification Scheme and use it as a custom authentication test with PAMs which allows a user to SSH into a remote computer. After building the proof of concept, we offer a security analysis of our system and identify potential attacks by adversaries and how we can prevent them.

Introduction and Motivation

Passwords are currently king in online authentication, but as we've discussed in class and as we've all seen in the news, passwords come with a lot of baggage. How to use them, store them, hide them, obfuscate them; there are clear right answers, but even with those right answers, serious mistakes are made. In 2016, over 3 billion passwords and user credentials were stolen – that's 95 stolen passwords per second [3]. Recently, Facebook announced it had been storing user passwords in plaintext. Facebook is no doubt a company well-versed on the concept of hashing and salting passwords, and probably has a handful of 6.857 graduates in their workforce. But when humans enter the equation and passwords are transmitted, serious and harmful mistakes can me made. What if there was a way to prove you knew the password without ever sending it to a server?

If users were able to prove they knew the password, and therefore were the correct user, they could be authenticated without ever sending this password. However, typical user passwords are often too short and non-random for zero-knowledge protocols, which offer the capability to prove your password without revealing it. In this paper, we discuss and implement an application of Zero-Knowledge Proofs [7] to remove password transmission from the equation.

This type of protocol has a handful of beneficial traits that encouraged us to investigate it. Never transmitting your password is clearly superior to sending it in any form. The non-interactive aspect allows these logins to happen in fewer steps; in an application like SSH, where remote servers may be distant and be separated by connections with high latency, removing two steps from a three-step process makes it much faster for the same latency.

Implementation

We took multiple steps to implement our Zero-Knowledge Proof PAM SSH Module. First, we coded our first pass of the Schnorr Interactive Zero-Knowledge Proof in Java. Once we agreed on the direction we wanted to go with our Zero-Knowledge Proof application, we made slight modifications to our Schnorr implementation to make it non-interactive. We used a Linux virtual machine on VMWare to simulate the remote machine to SSH into. Then we created a custom PAM that allowed or disallowed the SSH connection based on a query to our Schnorr implementation. Finally, once we added our custom module to the SSHD configuration file, our end result was a custom authentication protocol for SSHing into a remote machine.

Elliptical Curve Cryptography (ECC)

We used the ideas of Elliptical Curve Cryptography to implement the Schnorr Zero-Knowledge Proof Identification Scheme [6]. The general idea of ECC is perform calculations with points on an elliptical curve which creates a sort of one-way function. Adding and multiplying elliptical curve points is considered one-way because after performing these operations, it's very difficult to guess or compute which two points were operated on that resulted in the final point. Guessing the original two points becomes even harder the more and more operations are done.

This is the idea behind elliptical curve point multiplication. A point on the elliptical curve is added to itself a scalar number of times to produce a new point on the curve. Because the point successively adds to itself and wraps around the elliptical curve and falls in different places, it's virtually and computationally impossible to determine how many times the point was added to itself. This is how the secret and public key are related in the Schnorr Protocol: the public key is just the generator point G added to itself $\{\text{secret key}\}$ number of times.

`secp256k1` [8] defines the widely accepted parameters used in ECC. This is the current standard, and is used for Bitcoin's public key cryptography. These parameters have gained popularity in the last few years because the elliptical

curve was formulated in such a way that makes calculations computationally faster without comprimising security. Under `secp256k1`, the elliptical curve is defined as $y^2 = x^3 + ax^2 + b$, where $a = 0$ and $b = 7$. `secp256k1` gave us a jumping off point and defined some important constants, such as the generator point G, which is a known point on the elliptical curve. The order n of G and the cofactor h are also defined under `secp256k1`. All these parameters are very large numbers that don't seem particularly special to the human brain, but they were chosen in such a way that makes ECC efficient and secure.

Schnorr Interactive and Non-interactive Zero Knowledge Proof

The Schnoor Identification scheme [4] is an identification scheme based on Zero-Knowledge proofs, wherein the prover, who wants to be verified, shares no knowledge of the secret key whilst proving to the verifier that they indeed hold it.

Interactive Schnorr

The steps to the Schnorr Interactive Zero Knowledge Proof are outlined below. In this case, Bob is the Prover and Alice is the Verifier. The Prover and the Verifier exchange information in a sort of handshake protocol fashion. This protocol is summarized in Figure 1.

1. Bob chooses secret key a at random from $[1, n-1]$.
2. Bob computes and publishes public key $A = G \times [a]$.
3. Bob chooses v at random from $[1, n-1]$ and computes $V = G \times [v]$ and sends V to Alice.
4. Alice chooses challenge c at random from $[0, 2^{t-1}]$ where t is the bit length of the challenge. Alice sends c to Bob.
5. Bob computes $r = v - a \cdot c$ and sends it to Alice.
6. Alice performs the following checks:
   - Verifies A is a valid point on the curve (i.e. not the point at infinity)
   - Verifies $G \times [r] + A \times [c]$ is equal to the original V Bob sent

As we clearly see, Bob never had to share his secret key with Alice. Through ECC, Bob and Alice were able to perform calculations with only publicly available information to complete the identification process.

Non-Interactive Schnorr

The non-interactive version of the Schnorr protocol is very similar to the interactive version. The only difference is that the Prover (Bob) produces the challenge c instead of the Verifier (Alice). The challenge is defined as the hash of essentially all the publicly available information $c = H(G||V||A||UserID||OtherInfo)$. This allows the Prover to send one packet with all the necessary information and the Verifier can simply accept or reject immediately. We decided to move forward with the non-interactive version of the protocol because it was easier for us to transmit only one packet instead of support sustained communication between the Prover and the Verifier. However, as we discuss in the Security Analysis section, this can leave our system more vulnerable to adversaries and certain attacks.

The non-interactive version of Schnorr is still secure, under the Fiat−Shamir transformation [9]. The Fiat−Shamir transformation provides a way to take a multiple step, interactive proof and transform it into a non-interactive proof by

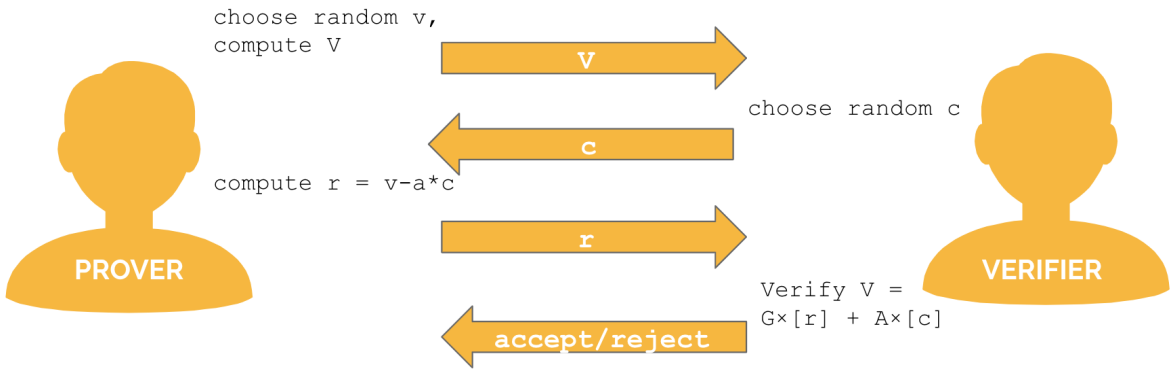

Figure 1: Schnorr Interactive Zero Knowledge Proof. Before the protocol begins, the Prover has already chosen a private key and computed and published the public key. The Prover and the Verifier send a series of messages back and forth during the protocol as the Prover attempts to authenticate him/herself.

replacing the challenge sent by the Verifier. The challenge in the Schnoor protocol is created with a cryptographic hash function of publicly available information, so the Verifier can check that the challenge the prover provides is indeed valid. The Fiat−Samir heuristic is secure under the Random Oracle Model, as the probability of the Prover making a correct response without actually following the protocol and without having the secret key is very, very low.

Outline of Code

We implemented the Non-interactive Schnorr Protocol in Java. We had three classes, `ECPoint.java`, `Prover.java`, and `Verifier.java`. Java's built in `BigInteger` package handled all the unusually large numbers and the computations on them. `ECPoint.java` handled all the ECC; each `ECPoint` had an x and y coordinate and could be added to another point, squared, or multiplied by a scalar. `Prover.java` generated a secret key and calculated the public key based on that. Then, the `Prover` randomly picked v and calculated V, comptued c, and sent V, c, and the public key to the `Verifier`. The `Verifier` computed the `ECPoint` produced by the computation $G \times [r] + A \times [c]$ and ensured that the `ECPoint`'s x and y coordinates matched V's x and y coordinates, respectively.

Unfortunately, the PAM discussed in the following section required the verifer code to be in Python. Instead of re-writing the whole protocol in Python, we decided to use `JPype`, which allowed us to call the Java verifier code from within a Python script. The Java code simply outputted `y/n` to a text file that the PAM could read from to determine if the Prover was authenticated or not.

Pluggable authentication module (PAM) [5]

We used PAM in order to customize the Linux SSHD process. This section briefly describes the existing PAM structure before going into our implementation of a custom module. All the code described here is available in our GitHub repository under a folder titled `pam_files`.

PAM Structure

The PAM structure consists of configuration files and modules. Configuration files give a high-level list of what modules are needed in order for an operation to succeed. For example, the `\etc\pam.d \sshd` configuration file

lists the modules run in order to test the validity of an SSH connection. Modules are shared object files (.so) that will output success or failure.

Modules are listed with groups and control flags. Group is one of four the PAM provides: auth, account, password, or session. A control flag indicates how its output value contributes to the success of the overall operation. Control flags consist of keywords such as `required`, `optional`, etc. Because we wanted to modify the SSH process, we modified the SSH configuration file by adding an *auth* module with the *required* control flag.

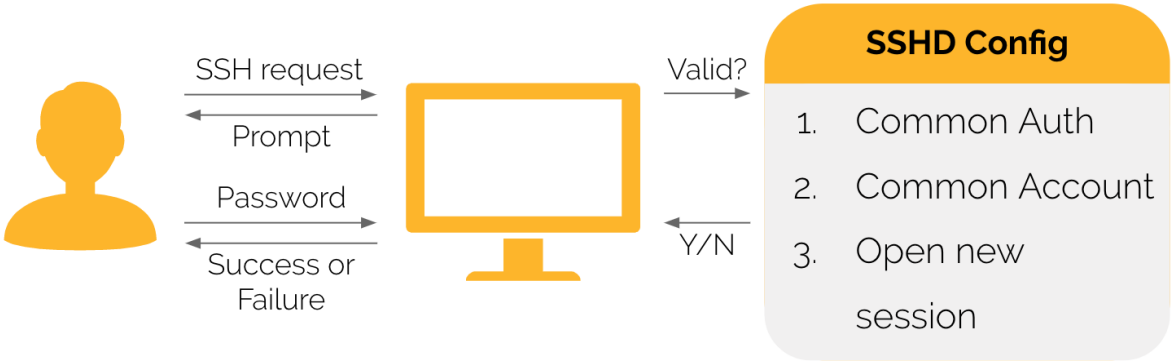

Figure 2: The existing pipeline for SSH with the built-in SSHD configuration file

Modified Configuration File

The existing SSH configuration file uses the built-in `common-auth` module, which will simply take the user's password and verify it. We loosened this module, by changing its control flag to be optional since we no longer wanted to verify a password. We then added our custom module above the `common-auth` module, like so: `auth pam_schnorr.so required`. In this way, the SSH connection would only occur if our custom module succeeded.

Custom Module [10]

Our custom module only affected the password verification process of authorization, specifically by changing it into a packet verification process. In the interest of keeping the first model simple, we required that 9 packet variables be passed in with commas as delimiters. These nine were the GenPoint, Public Key, V, r, n, and user ID with each elliptic curve point being broken down into its respective x, y components. After parsing, these 9 values were passed into a Python script as strings. As mentioned above, this script in turn called our Verifier in Java, which outputted acceptance or denial to a text file. The module would then process the text file, and appropriately return success or failure to the configuration file.

Setup

We edited or created five different files for our PAM system. The main changes include the SSHD configuration file (`\etc\pam.d \sshd`) and the custom module (`\lib\security \pam_schnorr.so`). In order to mimic our setup, it is necessary to replace several files in the existing PAM structure and restart the ssh system. A detailed overview of the PAM setup process can be found in our GitHub, at the `README.txt` file in the `pam_files` folder.

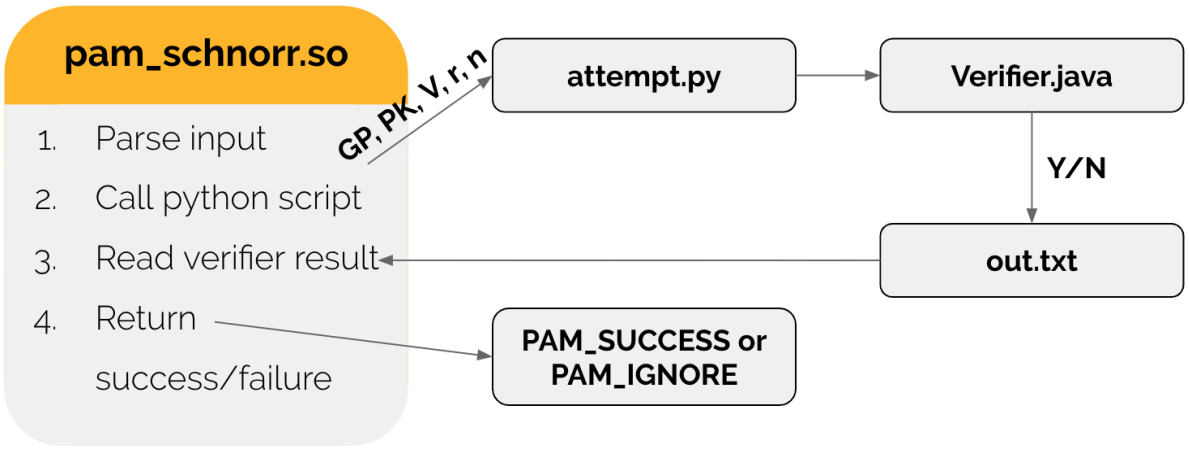

Figure 3: An overview of the custom `pam_schnorr.so` module. The success of the SSH connection attempt depends on the output of this module.

First Pass Product

As a proof of concept, we built out a simple GUI to simulate a user creating the input string for an SSH attempt. Once the user clicks the appropriate machine they wish to SSH into, the GUI outputs a string in the proper comma-delimited format that our module expects. The user can then paste this into their SSH query terminal. We have verified that the pipeline correctly rejects incorrect inputs while allowing proper ones. In the future, we would like to look into using Challenge Response Authentication with PAM to allow the user to input the 9 variables independently without worrying about string format. Moving forward we would also work to either shift the Java and python code into C or smooth the transition in some other fashion. Nonetheless we feel that for a first pass this is a good proof of concept and successfully shows an example of SSH authentication using zero knowledge proofs.

Security Analysis

User Roles

Provers

The group who are requesting authentication. They do this by computing a proof and sending that proof to a Verifier. If the proof is correct, they expect the Verifier to provide authentication. They are interested in the correctness of the solution, and the security provided through the protocol.

Verifiers

The group who are responsible for computing whether or not the proof they recieved is valid. This is being done to provide authentication to the Prover. Their roles are recieving the transmission, computationally verifying its accuracy, and returning the correct output. They are most invested in the correctness of the system.

Trusting 3rd Parties

Third party entities who may use the Verifiers to provide their own authentication. They have no active role in the system, and are mostly invested in the validity of the Verifiers.

Potential Vulnerabilities

There are several vulerabilities we forsee possibly being issues with the given system.

Replay Attacks

Because there is no handshake involved in the non-interactive version of the protocol, this system could be vulnerable to replay attacks; whereby an adversary can record what is sent from the Prover to the Verifier, and later use that same input to verify themselves. The copied message will be valid because the Verifier does not provide any of the information in the proof. This could be overcome by including a timestamp in the `OtherInfo` section of the challenge and transmission. The Verifier would then only accept packets within a certain time interval (say, 5 seconds) and reject any packet with a timestamp outside this grace period. However, this would still provide an adversary with a valid packet, albeit for a shorter window. For systems where a timestamp is not reasonable, this could become an issue.

Related Key Attacks

As we learned in lecture, related key attacks are situations where an adversary can observe multiple different signatures with different keys where the adversary doesn't originally know the value of the secret keys, but knows some mathematical relationship between the keys. With enough signature examples, an adversary can extract secret information from the signed messages. Morita et al [1] proved that the non-interactive Schnorr portocol was secure against a weak notion of RKA (wRKA), but insecure to the standard simple linear RKA. In their paper, Morita et al slightly modify the current Schnorr scheme to make it RKA secure, where an extra input (the recalculated value of the verification key) is added to the hash function. The exact details are provided in the paper, and we feel confident that we could implement this updated version of Schnorr in our system to prevent again related key attacks.

Malicious/Dishonest Verifier

The Interactive Schnorr Protocol is only a secure zero-knowledge protocol when we assume the Verifier is honest [2]. This means that we assume the Verifier is not malicious and is not trying to extract information from the Prover. A malicious Verifier might try to choose a challenge c in a non-random way, such that s/he could extract information about the Prover's secret key.

Because our system implements the non-interactive version of Schnorr, we luckily don't have to worry about this type of attack. Avoiding the need for an honest Verifier is a compelling reason to stick with the non-interactive version of the protocol. However, if we were to switch over to an interactive version, this is something we would have to consider.

Faster computation/Pre-computation

In a world with faster computing, discrete-log may not be as computationally difficult, allowing for the secret key to be recovered. Similarly, if primes are reused in a large-enough set of systems, pre-computation could be used to match a public key to its secret key. The solution to these problems for now is to select larger and larger primes, and

to sufficiently vary them. A truly random number generator is also essential. If v is repeated in two separate instances from the Prover, an adversary could observe both packets of information (V, c, r) and (V, c', r'), and this information can be used to compute the secret key.

Storing the Secret Key

If the secret key is not stored safely, the zero-knowledge aspect of the protocol no longer matters. Sufficiently storing the secret key is paramount to the security of the system. If the secret key is easily guessed or hackable, then obviously the enitre protocol is insecure and useless. Ideally, in a future iteration of our system, we would build out the "Password Manager" piece of the system that would effectively obfuscate the secret key. We can even impose a rule that the secret key must be changed every month or so, as our GUI provides a streamlined process for the user to input the other parameters the Verifier needs. Changing the secret key every so often would not be a burden on the user but would ensure the security of the secret key.

Conclusions

As the presence of technology in our daily lives grows, the notion of security in turn becomes increasingly important. We believe that the current reliance on and belief in passwords as a completely secure system can have the unintended effect of allowing users to become complacent with their digital security. As such, we were interested in exploring a newer, more secure method of user verification, zero knowledge passwords. Specifically, through implementing the Schnorr protocol of zero knowledge proofs, we were able to demonstrate a system in which a password never has to be sent online.

Having demonstrated the concept, we wanted to show its application to the universal task of SSH. We used Linux PAM to customize the authentication protocol for SSH. In this first pass, we created a full pipeline where, given a challenge, a user can prove their identity in order to be allowed into an SSH connection with the Linux machine. Although we used non-interactive Schnorr for our system, in the future, having familiarized ourselves further with the PAM structure, we will be able to implement an interactive system as well. In creating this basic product, we have shown how easily systems can move towards more secure protections in place of basic passwords, and hope that this work will persuade higher forms of security moving forward.

Acknowledgments

We'd like to thank our TA, Leo, for his guidance and help throughout this project. We'd also like to thank the 6.587 professors and the rest of the staff for a great semester.

Appendix: Code

Our GitHub repository containing the code for our Schnorr Non-Interactive Proof SSH Process is located at
`https://github.com/vonderhaar/6857-PasswordManager`

The `README.txt` file located in the `pam_files/` directory explains how to set up and run the files, which edits the SSH process for your machine.

References

[1] Hiraku Morita et al. *On the Security of the Schnorr Signature Scheme and DSA against Related-Key Attacks*. URL: https://eprint.iacr.org/2015/1135.pdf. (accessed: 05.15.2019).

[2] R. Gennaro et al. *Batching Schnorr Identification Scheme with Applications to Privacy-Preserving Authorization and Low-Bandwidth Communication Devices*. URL: http://www.ccs.neu.edu/home/koods/papers/gennaro04batching.pdf. (accessed: 05.15.2019).

[3] Justine Brown. *An average 95 passwords stolen per second in 2016, report says*. URL: https://www.ciodive.com/news/an-average-95-passwords-stolen-per-second-in-2016-report-says/435204/. (accessed: 05.15.2019).

[4] Feng Hao. *Schnorr Non-interactive Zero-Knowledge Proof*. 2017. URL: https://tools.ietf.org/html/rfc8235.

[5] *Linux-PAM*. URL: http://www.linux-pam.org/. (accessed: 05.15.2019).

[6] Saif Rehman. *Schnorr Non-interactive Zero Knowledge Proof for not so dummies*. URL: https://blog.goodaudience.com/schnorr-non-interactive-zero-knowledge-proof-for-not-so-dummies-16c70b40e0c8. (accessed: 05.15.2019).

[7] Ron Rivest and Yael Kalai. *Zero-Knowledge Proof Lecture Notes*. URL: https://courses.csail.mit.edu/6.857/2019/files/L22-Zero-Knowledge.pdf. (accessed: 05.15.2019).

[8] *Secp256k1*. URL: https://en.bitcoin.it/wiki/Secp256k1. (accessed: 05.15.2019).

[9] *What is the Fiat-Shamir transform?* URL: http://bristolcrypto.blogspot.com/2015/08/52-things-number-47-what-is-fiat-shamir.html. (accessed: 05.15.2019).

[10] *Writing Your First PAM Module*. URL: http://www.rkeene.org/projects/info/wiki/222. (accessed: 05.15.2019).