# WATSUP: Web Authentication without Sending or Storing User Passwords

Ryan Amos, Gregory Gundersen, Thomas Schaffner

## Abstract

ABSTRACT HERE

## 1 Introduction

Internet users must entrust private information with many different companies, but online security is extremely difficult. Even large and technically advanced companies have lost sensitive information to data breaches and malicious attacks. For example, Yahoo suffered several major breaches in the past few years. First, in 2013 attackers perpetrated the largest recorded data breach when they stole roughly one billion Yahoo accounts [19].

Then in 2014, a separate attack compromised roughly 500 million Yahoo accounts [8]. Neither of these breaches were discovered until 2016, meaning sensitive user information such as hashed passwords and security questions were stolen years before any user became aware.

These types of breaches are not unique to Yahoo [13, 14]. They exemplify a major issue with internet security: users must entrust their personal information to servers and companies that are not transparent, and sometimes not competent, in their implementation of security protocols. Most companies do not publish their full security practices, and even users who are responsible or knowledgeable about online security have no means to verify that their data is being handled correctly. Furthermore, the companies themselves are often unable to detect breaches promptly. As users register for an increasing number of services, the risks described above grow as well.

In spite of the difficulties and flaws of passwords, such as weak user passwords, password reuse, forgetting passwords, and constant breaches of improperly stored passwords, passwords are still unavoidable for users. Despite push from both users and security experts, alternatives rarely catch on, especially in web authentication [12]. Given that passwords have not been eliminated, we focus on improving the security of user passwords, rather than eliminating them entirely. We propose Web Authentication without Transmitting or Storing User Passwords (WATSUP), a new application-layer protocol that provides users with a transparent, consistent, and easy way to log into a number of different services without trusting any service to properly handle their login credentials. In this paper, we first discuss related work and its drawbacks; next, we propose the WATSUP protocol; third, we discuss a proof-of-concept implementation; and finally, we discuss the advantages and disadvantages of WATSUP and propose future work.

## 2 Background

### 2.1 Risks of using passwords

Verifying a user's identity is an important challenge in internet security. Passwords, despite their many drawbacks, are a critical and ubiquitous solution to the problem. Today, anyone who shops, banks, communicates, or socializes online creates multiple user accounts with a number of websites. But password usage has a number of risks, and existing solutions have disadvantages. The WATSUP protocol is designed to eliminate or mitigate as many of the most common risks mentioned below.

#### 2.1.1 Eavesdropping on plaintext

**Solution:** *Secure HTTP*
**Complication:** *Is not always supported*

The most straightforward risk is that a user's password is captured in transit as plaintext. The most popular solution to eavesdropping is to use HTTP

Secure (HTTPS), which tunnels an HTTP connection through a TLS connection. Unfortunately, while HTTPS is supported by most large companies, it is still not universally used [6]. In addition, many users may not realize that they should not use websites that do not support HTTPS.

#### 2.1.2 Server being compromised

**Solution:** *Salting and hashing*
**Complication:** *Requires trusting server*

Perhaps the biggest risk users face is that their passwords are compromised on a company's web server. As discussed in the introduction, these attacks have happened or been discovered as recently as last year, often at staggering scales. The most common methods to mitigate the effects of losing passwords is to salt and hash them. A cryptographic hash function is a one-way function for which it is easy to verify that an input is correct, but hard to extract the input from the output. A company should hash a new password before saving it in a database; to authenticate a user, it hashes the candidate password and compares it to the hashed password on record. "Salting" is adding random information to each password. Both of these methods help prevent rainbow table attacks in which precomputed tables of common password hashed with typical hashing algorithms are used to decode hashed passwords. They also help prevent an attack reusing a compromised password on another website. Unfortunately, many popular websites do not properly handle users data. For example, in its official statement, Yahoo only confirmed that the "vast majority" of passwords had been securely hashed before they were stolen [16], implying some (out of a billion) were not. In the 2012 LinkedIn data breach, LinkedIn failed to salt users passwords and only hashed them with the SHA-1 algorithm [10, 17], which is known to be insufficiently secure as early as 2005 [20].

#### 2.1.3 Phishing

**Solution:** *Hashing on hostname*
**Complication:** *Susceptible to rainbow table attacks*

Another common means of compromising a user's account is called "phishing." In phishing, a user is asked to submit their username and password to a malicious website that poses as a legitimate one, typically through email request to reset a password from a domain posing as the legitimate service. One solution to this problem is to hash the password with the website's hostname before sending it. If this is done consistently, phishing is prevented because the malicious website will have a different hostname. This allows for improved portability, since no secret data is stored, and no trusted third party is required [11]. However this method is still vulnerable to weak passwords and poor server salting. For example, an attacker who has compromised a database of passwords that has been salted with the hostname and then hashed can still create a rainbow table for that website. WATSUP's solution to this problem is to salt the base password with the hostname with the username. This prevents a rainbow table attack since the salt is unique for each user.

#### 2.1.4 Replay attack

**Solution:** *One-time passwords*
**Complication:** *Not typically implemented*

Another common attack vector is called a "replay attack." In a replay attack, an adversary intercepts the user's authentication credentials and simply retransmits them in a subsequent login to masquerade as the original user. To prevent this, each communication can use a nonce, which is a number that can only ever be used once. Typically, they are generated by a web server and sent to the user after an initial verification step. For example, Google provides dual authentication using nonces; but this service is only provided when the user logs in from an unknown device. WATSUP's goal is to use OTPs for every log in without additional overhead for the user.

#### 2.1.5 Password reuse

**Solution:** *Strong, unique passwords*
**Complication:** *Cognitive overhead*

Finally, passwords are insecure because remembering many strong passwords is hard, and users have a tendency to reuse passwords. Many systems have been designed to lessen the cognitive load for users. For example, password managers are popular tools to generate and save unique user passwords, reducing password reuse without increasing a user's cognitive load. However, these password managers are frequently insecure [15]. Additionally, they often rely on a trusted, non-transparent third-party or are not portable. Importantly, a password manager does not prevent phishing or replay attacks. WATSUP combines the ease of use of a password manager without storing the user's password, by providing a single, transparent version of source code, and still prevents phishing and replay attacks. Related work One solution to many of the above problems is Password Multiplier, which derives a site-specific password from a base password and the site's hostname. This uses SHA-1 as a key derivation function, and the hostname as a salt. Key derivation functions are functions that take the user's base password and produce multiple distinct high entropy keys by using a distinct salt. If one derived key is compromised, the base password and all other keys remain secure.[11] This does not protect against replay attacks on the same site. If everyone were to use this and the server failed to adequately salt and hash the derived keys, then everyone would be using the same salt. In this scenario, a rainbow table attack on a per-hostname basis would be reasonable, particularly against large databases.

Another similar solution is a client-server web authentication protocol known as Secure, Quick, Reliable, Login (SQRL). It performs web authentication on a "something you have" model. SQRL uses a randomly generated master password to provide strong, derived logins on a per-site basis, almost completely eliminating user passwords and making phishing difficult. On a login request, a link or QR code is provided for input to the SQRL app, which may be on a different device. The SQRL app then completes the login. As a side effect, this also protects against phishing [9] However, SQRL has a few issues. First, it requires its own protocol, and does not run over HTTP. This is a significant barrier to deployment. Second, it requires the rendering of QR codes. Finally, the code is implemented in assembly. While this was done so that the code is as clear as possible, e.g. the compiler cannot eliminate security-critical code, it makes incremental deployment difficult. In most respects, we argue that SQRL is simply more complicated than WATSUP without additional benefit.

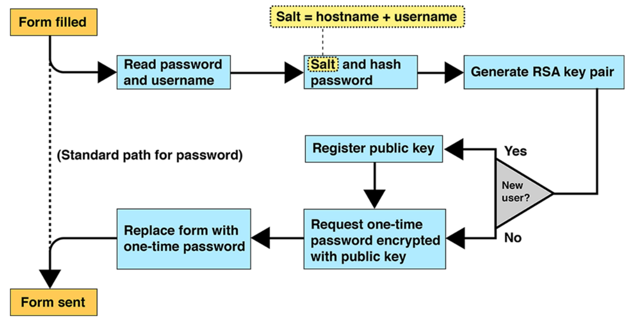


Figure 1: Protocol control flow

## 2.2 Implications

We argue that any solution to the web authentication problem must resolve the following issues:

- The solution should be portable. It should not require any data be stored or transmitted before it can be used on a new device. This means using passwords.
- Users should not have to trust servers.
- Users should be protected against replay and reuse attacks, even without HTTPS.
- The solution should allow for phased deployment by not interfering with standard login
- The solution should be as simple as possible for servers to implement, so that any developer can safely add it to their server, without risking user security

# 3 Design

The standard web login protocol requires the user to input their cleartext password and transmit it to the server, ideally via HTTPS. The WATSUP protocol derives a pseudorandom number generator (PRNG) seed from a key derivation function a user's base password with the hostname and username as a salt. While salting the hostname protects the user against password reuse, salting the username protects against poor server-side salting and hashing. The PRNG is used to generate an asymmetric key pair, which is used to prove identity by decrypting a server-generated nonce.

WATSUP generates a unique asymmetric key pair for each site. The key pair is generated with:

```
salt = SHA_256(SHA_256(hostname)
               | SHA_256(username))
bits[256] = PBKDF2(root_pass, salt,
                   1000, SHA_256)
prng = seed_ARC4(bits)
key_pair = rsa_key_gen(prng)
```

Upon registration, the user's public key is stored on the web server. When the user logs in, the server sends the user a cryptographic nonce, encrypted with the user's public key. The WATSUP client uses the generated private key to decrypt the nonce, which it sends to the server as proof of identity.

To support the WATSUP protocol, a server needs to implement the following specifications:

- On registration, the server stores the username and associated public key.
- On OTP request, generate a cryptographic nonce, associate the nonce with the requesting username, fetch the user's public key, use RSA to encrypt the nonce with the public key, and return the encrypted nonce.
- On authentication request, verify the received nonce matches the known nonce. If they match, mark the user as logged in. If not, reject the request.

# 4 Methods

The proof-of-concept implementation has two components: a back-end web server that is a stand-in for any web application that implements the WATSUP protocol, and an open-source browser extension that sandboxes the login form from the web page and implements the client-side of the WATSUP protocol.

## 4.1 Server

The server is responsible for generating cryptographically secure nonces, encrypting them with users' public keys before sending them to the client, and handling traditional user registration and login once the correct nonces are returned.

To generate the nonce, the server uses Python's `random` library function `SystemRandom` which is designed for cryptographic use. The function generates random numbers from information seeded by the operating system. Since the server runs on a UNIX-like operating system, `SystemRandom` queries `/dev/urandom` [4, 5].

To generate the RSA key pair the servers uses `cryptography`, an open-source Python package developed by the Python Cryptographic Authority [3]. The package primarily delegates to "backends" such as OpenSSL and CommonCrypto for the actual for platform-specific cryptographic algorithm implementations. The server uses the SHA-1 hash function for optimal asymmetric encryption padding; SHA-1 is acceptable here because adding padding does not hide sensitive information.

The back-end is written in Python 2.7 and uses the Flask web framework. The server is an instance of Gunicorn, a Python WSGI HTTP server. The server runs on a Heroku instance running Ubuntu 14.04.

## 4.2 Client

The client is implemented as a Google Chrome extension and therefore written in JavaScript, HTML, and CSS. While some portions of the code are Chrome-specific, other modern browsers provide the same functionality through similar APIs, and the codebase could be modified to support multiple browsers.

Importantly, Chrome sandboxes all browser extensions to prevent malicious websites from accessing privileged, extension-only operations; the concept is called the Content Security Policy [1]. The WATSUP client utilizes this security policy to keep user passwords safe. No other JavaScript program can access the extension's JavaScript program or HTML, both which would contain references to the userâĂŹs plaintext password.

For the protocol, first, the extension reads the username and password from their input fields and the hostname from Chrome's tabs API [2]. Then the program produces a âĂIJsaltâĂİ by separately hashing the username and hostname with SHA-256, and then rehashes the combination of these two hashes. This "salt" is used for generating a cryptographic key. While this information is deterministically generated, it still serves the purpose of a salt by preventing against rainbow table attacks. Next, the extension generates a cryptographic key from the base password and the salt. It uses Password-Based Key Derivation Function 2 (PBKDF2) for 1000 iterations using SHA-256 and the salt to generate 256 bits. This added computation is called "key stretching" and makes the password more difficult to crack.

Finally, the program generates the RSA key pair. The generated bits are used to seed an ARC4 PRNG, which is used to generate an RSA key, as provided by `cryptico` [18].

Upon user registration, the username and public key are sent to the server. Upon user login, the user sends the username to request a nonce. The server responds with a server-generated nonce, encrypted with the user's registered public key. The extension decrypts the nonce with the user's private key. The decrypted nonce is then sent to the server. Since the key pair is deterministically generated, the extension re-generates them upon every login. This ensures that the user's actual password is never saved or sent.

The code for both the server and client are open source and available here: `https://github.com/watsup-protocol`.

### 4.3 Deployment

Deployment of new technologies and protocols to the internet can be a complicated process, and adoption tends to be slow and spotty. We believe the first step would be to recruit a major browser to implement client-side WATSUP, while providing open source browser plugins for all major browsers. Then, we could recruit some major websites and web frameworks to adopt WATSUP. Due to the minimal work required to implement server-side WATSUP, and cross-compatibility with existing login services, any web service should be able to provide the service with ease.

## 5 Related Work

## 6 Conclusions

The WATSUP protocol provides usage guarantees for user passwords and reduces the number of points of possible exposure of user passwords. Because the WATSUP protocol never stores or transmits client passwords, users are guaranteed that their cleartext passwords will not be stolen during transit or authentication or from the server. Additionally, this guarantees that remote servers are unable to store cleartext or weakly protected passwords themselves. Users of the WATSUP protocol no longer need to trust a company's security systems to protect their passwords since servers never see cleartext passwords. This mitigates the effect of a data breach.

However, the WATSUP protocol as described in this paper still suffers from some existing security issues. Current common implementations of asymmetric encryption schemes are vulnerable to offline attacks [7]. Additionally, if the user chooses a weak base password, the base password may be susceptible to offline dictionary attacks. Both of these attacks are only possible if an attacker obtains a nonce encrypted with a client's public key, or the attacker obtains the public key itself. Even with alternate asymmetric encryption algorithms, the attacker can still perform an offline dictionary attack if they also capture the plaintext nonce; the attack would perform a standard dictionary attack but then generate a public key for each candidate password using the WATSUP's protocol. If any public key decrypts the encrypted nonce and results in the unencrypted nonce, the attack knows the correct password. Alternately, with current encryption schemes, man-in-the-middle attacks are still possible with the WATSUP protocol. It is worth noting that these weaknesses do not expose the client's base password, and would only allow an attacker to impersonate the client when communicating with a single server under a single username. These weaknesses can be eliminated by using HTTPS, but the user still has some guarantees using HTTP.

Another weakness is a Denial of Service attack against the login protocol. An attacker could rapidly request nonces for a target user. Depending on the server implementation, a request may invalidate previous nonces, so the target may be unable to log in.

The WATSUP protocol also does not protect a user from themselves. In order for the WATSUP protocol to provide portable access to online content, a client's private key is reproducible using the client's username and password for a given host. Therefore, if a client loses their password themselves, through some social engineering attack for example, their accounts will still be vulnerable. This also means that online dictionary attacks are still a threat if clients employ weak base passwords. We do not propose WATSUP as a replacement for good password hygiene. Users should still use good practices, such as different passwords on each site and using high entropy passwords.

As discussed in the conclusion, modern asymmetric encryption schemes are not entirely secure. We recommend development and implementation of stronger asymmetric encryption schemes regardless of which authentication model is widely implemented. We also recommend extending the WATSUP protocol. Under the current implementation of WATSUP, users are not able to verify the identity of a server. Extending the protocol to allow for clients and servers to authenticate each other would be beneficial to users. If a client can verify the identity of a server, it can identify insecure situations and refuse to transmit sensitive data. Additionally, the use of HTTPS to secure all communication channels should be enforced whenever possible.

### 6.1 Future Work

Before WATSUP is ready for public use, it needs to be improved and thoroughly vetted by security experts. A major issue that needs to be fixed is the fact that the protocol relies on the server to produce a cryptographic nonce. Our security model does not allow for trust of the server. Therefore, we propose using POSIX time in milliseconds as a nonce. As long as the ping is greater than one millisecond, and the server is correctly providing the time, this nonce will be unique, and hence adequate for cryptographic use. The client can then audit the server's compliance by ensuring the client's POSIX time and the server's POSIX time match within some generous delta (e.g. 1 day). This could cause an issue if the client's clock is not accurate.

Another pathway that could be explored is that of mutual authentication—a trust-on-first-use system could be put in place. This would provide server authenticity guarantees to the user, which could be critical in the event of HTTPS PKI compromise. To solve this, one model is that the user saves the server's public key, but this means trust will need to be reestablished on each new device. Another model is that the user signs the server's public key, which the server saves and reissues to the user on each connection, but this makes revocation in the event of server compromise difficult or impossible. In either model, the server would perform authentication similar to the client authentication.

As previously discussed, WATSUP is vulnerable to offline dictionary attacks against the base password, since attackers can observe both the encrypted nonce and decrypted nonce. Additionally, properties of asymmetric encryption mean that access to just the encrypted nonce is sufficient for an offline attack. With better cryptography, the WATSUP protocol may be able to avoid these issues by never revealing the encrypted nonce in the first place. Tunneling WATSUP through a secure channel, such as HTTPS, already solves this. However, in situations where HTTPS is not available, it is still important to try to protect users. Any solution needs to follow the core ideas of WATSUP: it must be simple for the server to implement, and it must not rely on trusting the server.

## References

[1] Google, 2017. URL `https://developer.chrome.com/extensions/contentSecurityPolicy`. [Online; accessed 2017-01-12].

[2] Google, 2017. URL `https://developer.chrome.com/extensions/tabs`. [Online; accessed 2017-01-12].

[3] Python cryptographic authority, 2017. URL `https://github.com/pyca`. [Online; accessed 2017-01-12].

[4] Python 2.7.13 documentation, 2017. URL `https://docs.python.org/2/library/random.html#random.SystemRandom`. [Online; accessed 2017-01-12].

[5] Python 2.7.13 documentation, 2017. URL `https://docs.python.org/2/library/os.html#os.urandom`. [Online; accessed 2017-01-12].

[6] J. Aas. Progress towards 100 URL `https://letsencrypt.org/2016/06/22/https-progress-june-2016.html`. [Online; accessed 2017-01-12].

[7] M. Blumenthal. Encryption: Strengths and weaknesses of public-key cryptography. *CSRS 2007*, page 1, 2007.

[8] S. Fiegerman. Yahoo says 500 million accounts stolen, 2016. URL `http://money.cnn.com/2016/09/22/technology/`

yahoo-data-breach/. [Online; accessed 2017-01-12].

[9] S. Gibson. Secure quick reliable login, 2015. URL https://www.grc.com/sqrl/sqrl.htm. [Online; accessed 2017-01-12].

[10] R. Hackett. Linkedin lost 167 million account credentials in data breach, 2012. URL http://fortune.com/2016/05/18/linkedin-data-breach-email-password/. [Online; accessed 2017-01-12].

[11] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web*, pages 471–479. ACM, 2005.

[12] C. Herley and P. Van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36, 2012.

[13] T. Hunt. have i been pwned?, 2016. URL https://haveibeenpwned.com/. [Online; accessed 2017-01-12].

[14] O. v. K. Igal Tabachnik. Plain text offenders, 2016. URL http://plaintextoffenders.com/. [Online; accessed 2017-01-12].

[15] Z. Li, W. He, D. Akhawe, and D. Song. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 465–479, 2014.

[16] B. Lord. An important message about yahoo user security, 2016. URL https://yahoo.tumblr.com/post/150781911849. [Online; accessed 2017-01-12].

[17] N. Perlroth. Lax security at linkedin is laid bare, 2012. URL https://nyti.ms/1AgFElO. [Online; accessed 2017-01-12].

[18] R. Terrell. Cryptico, 2017. URL https://github.com/wwwtyro/cryptico. [Online; accessed 2017-01-12].

[19] S. Thielman. Yahoo hack: 1bn accounts compromised by biggest data breach in history, 2016. URL https://www.theguardian.com/technology/2016/dec/14/yahoo-hack-security-of-one-billion-accounts-breached. [Online; accessed 2017-01-12].

[20] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full sha-1. In *Annual International Cryptology Conference*, pages 17–36. Springer, 2005.