One-Time Password Access to any Server without Changing the Server

Dinei Florêncio and Cormac Herley
dinei@microsoft.com, c.herley@ieee.org
Microsoft Research, One Microsoft Way, Redmond, WA

Abstract. In this paper we describe a service that allows users one-time password access to any web account, without any change to the server, without changing anything on the client, and without storing user credentials in-the-cloud. The user pre-encrypts his password using an assigned set of keys and these encryptions are sent as one-time passwords to his cell phone or carried. To login he merely enters one of the encryptions as prompted, and the URRSA service decrypts before forwarding to the login server. Since credentials are not stored (the service merely decrypts and forwards) it has no need to authenticate users. Thus, while the user must trust the service, there are no additional passwords or secrets to remember. Since our system requires no server changes it can be used on a trust-appropriate basis: the user can login normally from trusted machines, but when roaming use one-time passwords. No installation of any software or alteration of any settings is required at the untrusted machine: the user merely requires access to a browser address bar.

Keywords: passwords, one-time passwords, authentication, replay resistance

1 Introduction

Users increasingly find themselves in the position of having to enter sensitive information on untrusted machines. As access to more and more services is pushed online, the range of sensitive information that a user must protect grows with time. Passwords are the most obvious example. Email, bank and brokerage accounts, employee benefits sites, dating and social networking sites almost universally allow password protected access to services. A user who logs in to any such account from an untrusted machine runs the risk that a keylogger will capture the password and allow unauthorized access. In addition, the number of machines that must be regarded as untrusted also grows. Most obviously, any machine at an internet café or kiosk must be assumed suspect. But additionally a user's own home computer can easily be infected with spyware.

The problem we address is to enable a user to login from a machine that is untrusted. For simplicity we will assume the worst: everything the user does on such a machine is observed and logged. Everything typed, everything that appears on the screen, and all of the network traffic is captured and is available to an attacker. Nonetheless we want to be able to login to password protected accounts from such a machine, without risking catastrophic loss of data. While there are widely varying estimates of the dollar size of the fraud problem that password stealing causes [20] the fear and confusion is very real. We assume that preventing passwords from falling into the wrong hands is more important than protecting the rest of the data from a session. Thus we are not protecting the privacy of data, and we do not prevent session hijacking.

Our approach for passwords to a particular account will be to generate a series of one-time passwords that can be used to login. The actual mechanism we employ will be for the user to navigate to a webserver that will act as a Man-In-The-Middle (MITM). We call the system URRSA: Universal Replay-Resistant Secure Authentication. The one-time data will be provided to the URRSA server, which then performs a decryption and substitution: replacing the one-time password typed at the suspect machine with the true password forwarded to the login server. In this way the sensitive information is not typed at the untrusted machine, and neither is it displayed or downloaded to the compromised environment; nor is it stored at the URRSA server.

We make several requirements of the solution in order to be useful:

- No change to existing login server.
- No change to the browser or client software environment. We do not assume that the roaming user has installation privileges. We do not require the user to change the browser proxy settings. Requiring users to alter browser proxy settings we believe makes the User Experience of Impostor [27] and KLASSP [15] unworkable for a realistic deployment.
- No storage of credentials in the cloud: this removes the single point of attack that such a server would represent.
- No authentication of the user to the service: if we try to authenticate with a password we are back where we started. The alternatives, such as smartcards, greatly increase the complexity of the service and the burden on the user.

Our main goal in this paper is to describe the technology and give sufficient detail to allow implementation. However, it is legitimate to question whether users will trust such a system. The answer is obviously dependent on who runs it. There are two deployment scenarios that don't involve trusting a third party. The first is that the login server runs the service. For example, using URRSA, PayPal or Fidelity might offer OTP access to those clients who desire it *without altering their current authentication process.* The second is that the user self-hosts the service on a machine that he controls. Obviously this solution requires that the user maintain a server and a fixed IP address or a domain name; so this is possible only for a small minority of users. Both of these deployment models are potentially useful, and get around the issue of trust by having the "third party" be one of the existing two parties. The final model, and potentially most useful, is of URRSA offered as a web-service hosted by a third party. The success of online financial management sites such as `www.yodlee.com`, `www.mint.com` and `www.wesabe.com` demonstrates that at least some users will trust such a service. At `Yodlee`, for example, users give the service passwords to their bank and credit-card accounts so that it can daily update bill and payment information.

In the next section we review related work. In Section 3 we show how any account can be transformed into a one-time password account without having to change the server or the browser. We review implementation details in Section 4. Section 5 examines attacks and Section 6 evaluates our deployment.

2 Related Work

2.1 Coping Strategies and Simple Tricks to Evade Spyware

Sometimes coping strategies can be enough to evade a keylogger. Herley and Florêncio [11] describe a simple trick that users can employ to confound keyloggers by obfuscating their passwords. By interspersing the legitimate password characters with random characters typed outside the password field, the technique is able to confuse most existing keyloggers. While useful, this is not a durable solution, as keyloggers could be easily modified to capture enough additional information to retrieve the actual password.

Another technique that can be used to authenticate without explicitly typing passwords involves storing the password or equivalent information in a bookmark, and accessing it from a standard web browser. A flexible way of achieving that is based on the use of *bookmarklets.*

These are small JavaScript programs that are stored as bookmarks. JavaScript is flexible enough that it can be even used to hash a general password, and generate site-specific passwords [1]. Storing the password (or equivalent information) in a URL link, bookmarklets avoid the need to type the target site password. Access could be achieved when roaming, by storing the bookmarklets on a USB drive, for example. Nevertheless, this is not necessarily safe. If its use were to become widespread, hackers could attack the bookmarklets directly. If the file containing the bookmarklets is copied when inserting the USB drive, every single password on the drive could be compromised (thus making the user less secure not more). It would make matters worse not better if checking a `hotmail` account exposed the `BankOfAmerica` credentials stored on the same device to be exposed.

Another group of solutions involves having the web site use some authentication method other than simply typing passwords - sometimes in combination with some typing. Examples include on-screen keyboards, two-factor authentication, challenge-responses systems, and many others. These usually apply only to the site that adopts that particular mechanism, and require a major change in the server and User Experience. Rather than have users key their passwords some web sites have experimented with on-screen keyboards as a method of secure data entry. These schemes can be attacked by having the keylogger do a screen capture at each mouse click event. An interesting work by Tan *et al.* [31] addresses the question of minimizing the chances that a password entered using an on-screen keyboard is captured by an observer. This work addresses the "shoulder surfing" risk rather than the risk that the machine itself is running spyware, but has interesting analysis of the usability of various alternative password entering mechanisms.

2.2 Challenge Response Mechanisms

The use of challenge response has been explored as a means of achieving resistance against replay attacks when the user must login from an untrusted environment. Cheswick [12] examines on a higher level the use of Challenge-Response authentication mechanisms to evade spyware. The advantage of such systems is that a spy who observes a successful login session cannot perform a replay attack: the challenge will be different for each event and observing a single response helps the attacker very little. Cheswick reviews a number of approaches from the point of view of usability. Each of these schemes would require changes at the server.

Pering *et al.* [28] explore the use of a series of the user's own images uploaded in advance. The user is then authenticated by successfully responding to a series of challenges, which essentially involve picking his images from random images. Another image based scheme is proposed by Weinshall [32]. To avoid an image-similarity attack, the images are assigned, rather than uploaded. Furthermore, to reduce the amount of information given out with each authentication session, the user is not directly asked which images are his. Instead, the image set memberships are used to select a certain path on an image mosaic, with the user providing only a code that depends on the path's endpoint. It is pointed out by Golle and Wagner [18] that observing as few as six logins of this scheme can allow an attacker to determine the secret. Coskun and Herley [13] show that a challenge response scheme that relies on the users memory and calculating ability alone is almost certainly vulnerable to brute-force attack.

2.3 Proxy-based systems

Four works that directly address authentication from untrusted machines are Impostor [27], that of Wu *et al.* [25], Delegate [21], and KLASSP [15]. All use a proxy to intervene.

The Impostor [27] system of Pashalidis and Mitchell, is a password management system where roaming users can access their credentials. Rather than have users authenticate themselves by typing a master password (as is the case for [17]), a challenge response authentication is used. The user is assigned a large string that forms the secret. When requesting access the user is challenged to provide characters from randomly selected positions in the string, and is authenticated only if she responds correctly. In this way the user reveals only a small portion of the secret string to any compromised machine. A replay attack is difficult, since the challenge positions change each time the user contacts the proxy. Nonetheless, Impostor potentially protects strong secrets with weak ones. If strong (*e.g.* 60 bit) passwords are stored with the system an extension of the three character challenge originally proposed would be necessary (as shown in [13] such a scheme is vulnerable to an attacker who observes several logins). Impostor runs as a HTTP proxy, and the user must direct their browser to the proxy. Wu *et al.* [25] sketch a similar architecture where a proxy stores credentials; the proxy delivers a challenge which must be answered by SMS to authenticate the user.

Another proxy-based system which stores users credentials is Delegate of Jammalamadaka *et al.* [21]. Like Impostor, they store the passwords in the cloud, and act as a proxy to serve as intermediary between the server and the untrusted terminal. Credentials are filled by the proxy in web requests as they are forwarded to the login server. A cell phone is used for the user to explicitly authorize credential insertion when necessary. This requires, of course, that the user has cell reception. By contrast our system has no such requirement. An additional feature is that Delegate uses a rule-based hierarchy to request additional authentication whenever a sensitive operation is requested. This can be used to reduce the risks of a session hijacking (e.g., by requesting additional authentication when a money transfer is requested) as well as to remove sensitive information (e.g., account balances) from web pages provided by the server. These rules are generally to be provided by security experts, or learned from the user on a previous interactive session from a safe terminal.

The KLASSP proxy [15] of Florêncio and Herley also functions as a MITM proxy for communication with the login server. The user enters a *mapped* password $M(pwd)$ on the untrusted machine, and the proxy *unmaps* before forwarding $M^{-1}M(pwd)$ to the login server. The mapping $M()$ has, of course, been agreed in advance between the user and the proxy and serves as a shared secret. KLASSP suggests two broad directions for mappings. In the first the user obfuscates the password by entering either password keys or random keys in response to prompts from the proxy (in a variation on [28] the prompts are a series of images, where the user’s personal images act as sentinels to signal a true password character). As with Impostor, this technique protects strong secrets with weak ones. In the second the user encrypts the password using a large and cumbersome encryption table.

Impostor and KLASSP have in common that they are implemented as HTTP proxies. This means that the user must force the browser on the untrusted machine to use the proxy. This is inconvenient and is not always possible (*e.g.* the user may not have permissions). In Internet Explorer this requires editing Tools, Internet Options, Connections, LAN settings, un-checking “Automatically detect settings”, checking “Use a proxy server”, entering the IP address and port number, clicking “Advanced options” and checking “Apply same proxy for all protocols.” Further the settings must be undone when the user leaves; if this is neglected or forgotten there is a risk that the next user of the machine has his traffic routed through the proxy also.

2.4 One-Time Passwords and S/Key

Several one-time password systems have been proposed that limit the attacker’s ability to exploit any information he obtains. Notable among them is S/Key [22, 29] which generates a series of passwords by iteratively taking a cryptographic hash of a secret key. At each login the server verifies that the hash of what the user presents is the previously used password. Since each of the passwords is used only once it is of no use to an attacker. A further advantage is that the server need store only the previously used password. Thus even the database at the server contains nothing useful to the attacker. A popular implementation of S/Key for Unix is described by Haller [19]. The user carries a list of OTP's; generally these are sequences of short English words, so the user must type a total of about 20-24 characters to authenticate. An alternative one-time password system is OTPW by Kuhn [2]. Instead of being generated from a single secret (as with S/Key) here the passwords are independently chosen random secrets, and the hash of each is stored on the server.

Mannan and van Oorschot [24] describe MP-Auth: a system that uses a trusted mobile device such as a PDA or smart-phone to enter the password. The device encrypts the password using the end server's public key before passing it to the untrusted terminal. MP-Auth has the advantage of not requiring (as we do) that the user trust a proxy, but does not work with existing login servers, and requires a channel (such as bluetooth) between the trusted device the untrusted machine.

SecurID from RSA [3] gives a user a password that evolves over time, so that each password has a lifetime of only a minute or so. This solution requires that the user be issued with a physical device that generates the password. This solution requires considerable infrastructure change on the server side, which has limited its use. However SecurID has the advantage of being immune to OTP stealing attacks (see Section 5.3).

2.5 In-the-cloud Password Managers

One sub category of challenge response system is worth a separate note: in-the-cloud password management systems. These systems store the sensitive information at a server in-the-cloud, and have the server deliver the sensitive information directly to the desired destination on the user's behalf. An early example is [17]. Storing all this sensitive information in the server provides a new vulnerability. Indeed, if an attacker gains access to the user's account at this server, it would have access not only to the information that the user typed, but to any other information stored there as well. Further, a server storing hundreds or thousands of users sensitive information can itself become a target for attacks.

An early in-the-cloud example, proposed by Gaber *et al.* [17], used a master password when a browser session was initiated to access a web proxy, and unique domain-specific passwords were used for other web sites. Since users authenticated themselves by typing the master password, this clearly offers no defence against keyloggers. The same is true of other in-the-cloud systems such as Passport, where the user authenticates himself using a master password, or `www.clipperz.com` where a passphrase is used.

2.6 Relation to our service

One-time Passwords offer well understood security enhancements over existing password systems. Our proposed scheme gets the excellent protection enjoyed by users of existing OTP systems [22, 19, 2, 3] to all users. We wish to be clear that we will have the same security and usability questions that arise with other OTP systems; *e.g.* from a usability standpoint the user must keep the OTP list safely, and, like most OTP systems, session hijacking is still possible. The advantages we offer over previous approaches is that we give OTP protection *without changes to the existing server.* A consequence is that users can have trust-appropriate authentication. When using a trusted machine they can continue to login as before. However when they decide the circumstances demand they can use one-time passwords to access the account. When an OTP list is generated it is sent by SMS text message to the user's cellphone, but users who prefer may print and carry a hardcopy OTP list. We view the cellphone as preferable for a number of reasons. It represents a device that the user generally carries anyway. The user is less likely to lose or misplace a cellphone than a hardcopy OTP list. Finally, by using an out of band channel like SMS to send OTP's to the user new OTP lists can be generated without requiring the user to return to a trusted location (see Section 4.3).

An advantage of the system we propose with respect to the proxy-based systems Impostor and KLASSP [27, 15] is that it is implemented as a MITM web service rather than a HTTP proxy. Users do not need to change the proxy settings on the browser before they begin and undo them when they are done. Thus the burden is much lower than with [15] or [27]. Further the service is involved only for the duration of the connection to a password protected account. For example an Impostor or KLASSP user during a one hour session might change the proxy settings at the beginning and undo them at the end. If he visited several password protected accounts, but also news and information sites the proxy would handle all of the traffic for the entire hour. With our MITM service implementation the URRSA server is involved only from login to logout on each password protected account. The entire traffic for `BankOfAmerica` would flow through the service, but none of the general browsing traffic would. Thus the load on the service is greatly reduced (in comparison with Impostor or KLASSP) and privacy is enhanced. Delegate [21] does not explain how its proxy mechanism works. We assume, since it makes no mention of the crucial processing of the request-response stream that is the heart of our system (Section 4.2), that it is also implemented as a HTTP proxy.

The URRSA service can be seen as a descendant of KLASSP [15], where the mapping $M()$ becomes a true encryption of the password, and the proxy is a reverse proxy. Independently, and after, one of the authors of Impostor also developed and deployed a similar system [4]. This appears to be based on a reverse proxy similar to CGIProxy rather than the scheme we describe in Section 4 but is similar in many other respects to our service.

3 Method

URRSA provides a service that allows users to access a website requiring authentication, without having to type the actual password in the clear. The only time the actual password is typed is during the registration, which is done in advance from a safe location. At registration, the user receives versions of the true password, each encrypted with a different key. The service will decrypt using any of the keys only once, so effectively the user receives One Time Passwords each of which can be used once to access the registered site from untrusted locations. Our approach does not store any passwords at the URRSA server. The server needs to store only the encryption keys used, not the actual password. By doing this we remove the information stored with the service as a vulnerability: the keys have no value to the attacker without the corresponding OTPs (and if an attacker has the OTPs, he doesn't need to steal the keys, just use the service). More importantly, we remove the need to authenticate the user. We wish to be clear however that, while the service does not store user passwords, the user must still trust the service.

3.1 Mapping Strategy

In theory passwords can contain upper and lower case letters, digits and any of a few dozen special characters. While in practice we know that the majority of users seldom use extended characters [14] we must nonetheless support all possibilities. Letters and digits give 62 characters and we allow for 66 special characters, for a total of 128 possible characters. Call this set $\mathbf{C}$. An obvious way of encoding the password characters would be to use a simple permutation code that maps $\mathbf{C}$ to itself, and to apply this to all characters independently. Thus a length N password (*i.e.* in $\mathbf{C}^N$) would be mapped to another password in $\mathbf{C}^N$. There are a few problems with this approach. First, permutation codes leak information when two characters of the original password are the same. More significantly we have the following complications:

– Confusion sets: certain characters such as the number "zero" and the letter "O", or the number "one", the upper case "I" and lower case "L" can be hard to tell apart when context is removed.
– SMS restrictions: certain characters (*e.g.* []{}) cannot be sent by SMS (see *e.g.* [5]).
– Unfamiliar keyboards: layout for the position of special characters varies greatly on international keyboards. Some characters requiring meta-keys (*e.g.* Shift, Alt, Alt-Gr *etc*) can be very hard to find.

It is important to exclude confusion sets. This is especially the case if the user carries the OTP list on his cell phone, since we have no control whatever over the fonts in which the OTP will be displayed. We require that every password map to an OTP that is unambiguously readable on any display. This rules out "O" and "0" *etc.* The set of characters that cannot be sent by SMS must be excluded from any mapping we produce. Since the phone will merely receive and display we have no opportunity to perform any mappings there. Finally, even common special characters such as "@" can be hard to find on an unfamiliar keyboard (*e.g.* on many keyboards it requires pressing the Alt-Gr key which often causes confusion since Alt-Gr does not exist on US keyboards). The problem is compounded when the meta keys such as SHIFT, Alt, Alt-Gr, Esc *etc* are labeled in a language or alphabet unfamiliar to the user. While we wish to support the minority of users who have unusual characters in their passwords we do not wish to force a user with a simple password such as "Snoopy2" to search for characters like "%" and "¿" on the keyboard in a Chinese internet café.

To avoid any and all confusion, we restrict the output OTPs to use only capital letters and digits, not including the above mentioned characters: "0", "O", "I", and "1". Therefore the valid characters are easily identifiable: ABCDEFGHJKLMNPQRSTUVWXYZ23456789. Call this set **D**. This gives us 32 characters, enough to carry 5 bits of information per character.

So we have an input password with N characters drawn from an alphabet of 128 symbols (*i.e.* the set **C**). We wish to map to an OTP drawn from an alphabet of 32 symbols (*i.e.* the set **D**). Clearly the OTP must be longer than the input password. We transform the input password to a string of $7N$ bits, and encrypt those bits using the one time encryption pad. We then map the result (5-bits at time) to a password OTPi with M symbols drawn from an alphabet of only 32 symbols. Clearly, OTPi will have $M = ceil(7N/5)$ characters. Thus the procedure maps a password from $\mathbf{C}^N$ to an OTP from $\mathbf{D}^M$. For example the 9 character input password "{Qp#oL{4s" might map to the 13 character OTP "RM8BQ47AAKW3U." The OTP contains only characters that are unambiguously readable on any display and easily found on any keyboard. We then repeat the process with different encryption keys to produce each of the desired OTPs. For decoding, we follow the reverse procedure. Pseudo-code for this encoding is given in Section A.1. To decrypt the password, the URRSA server needs only to know which encryption key was used. The url and userID pair allows this to be determined. After receiving this information, the server checks which was the last key used and informs the user which OTP to use next. This information is all that is required to tell the Service which key to use (see Section 4.2), and to guarantee that service and User are in sync. Since the service merely decrypts and forwards no authentication of the user with the service is required.

As described each input password of a length N would map to an OTP of length $M = ceil(7N/5)$. However, since we know that a majority of users choose weak passwords [14] this is actually somewhat wasteful. The two passwords "snoopy" and "G(r!e9" will map to the same length. If we Huffman encode [30] the plaintext password before encryption and mapping we can reduce the length of the average OTP that must be typed. Huffman decoding is done at login time. Note that we have not weakened user passwords in so doing; we have merely ensured that weak passwords from $\mathbf{C}^N$ will be mapped to the shortest possible strong password. It is also worth noting that password strength does not necessarily increase security when password stealing attacks such as phishing and keylogging are the main threats [16].

3.2 User Experience

The user merely navigates to `http://{URRSA}/OTPLogin` login page at the webserver and enters first the url and userID of the account he wishes to access (this allows the proxy to retrieve the keys used to generate that user's one-time passwords). The user then enters the k-th OTP from his OTP list, allowing the server to decrypt and temporarily store the true password. The user's browser is directed to open `https://{URRSA_1}` (which directs our server to fetch the registered login page). The user need type nothing further and merely clicks the submit button and login proceeds. Observe that the URRSA service does not authenticate the user. It has no need to, since it does not know any of the user's passwords and merely decrypts and forwards.

If we compare the user experience between logging in directly (*i.e.* navigating directly to the login server and risking a keylogger) and using our service we find as follows (we will use PayPal as an example). To go directly to the login server the user types `https://www.paypal.com` in the address bar and then his userID and password and clicks submit. To login using our service the user types `http://{URRSA}/OTPLogin` in the address bar, then types `www.paypal.com` and his userID at the loaded page and submits. A new page loads on which he enters the requested OTP and submits. Finally, when the `https://{URRSA_1}` page loads (which will be a copy of the `https://www.paypal.com` just loaded by the service) he clicks submit one more time. Thus it can be seen that the additional burden on the user is not very great: the user has one additional URL to type, and two additional clicks. The sequence of events is detailed in Section A.2.

3.3 Acting as a MITM Webservice

The MITM service that URRSA performs can be regarded as a reverse proxy [23]. Conceptually (if we dealt only with static HTML pages) a reverse proxy works by fetching a first document for the client and translating all links therein to again go through the proxy. For example if the client wants `https://www.abc.com/foo` it would instead ask for `https://{URRSA_1}/foo`. The proxy receives the request and forwards to the server at `www.abc.com`; the server delivers the request to the proxy, which passes it back to the client browser. This is not to be confused with a HTTP proxy, where the browser is configured to direct all traffic to the proxy [23]. Reverse proxies are also sometimes known as CGIproxies after a particularly popular family of implementations [6]. While conceptually simple, reverse proxying modern web-sites is a complex task. Examples of reverse proxy can be seen at any of a number of anonymizing web proxies. However most anonymizing proxies offer a somewhat brittle experience [26]. In particular dynamically generated links are often handled incorrectly, and missing images and broken links are a common experience. In addition many are unable to handle SSL traffic successfully, certificate errors are common, and cookies do not always get correctly assigned. We are unaware of a single anonymizing reverse proxy that is robust enough to handle the complicated request/response stream that occurs during an authentication.

We are able to simplify the problem by attempting considerable less generality than anonymizing reverse proxies offer. Rather than reverse proxy for any possible domain, and all the links therein we seek to handle only the limited number of domains and sub-domains encountered during login at a site. A login server generally has links to fewer external domains than a conventional site (*e.g.* when logging into `www.PayPal.com` a user sees essentially only content from the `PayPal` and `PayPalObjects` domains while loading `www.nytimes.com` involves as many as ten distinct domains). So our task will be to translate only for the domain(s) to which the user is authenticating and not for a plurality of sites.

4 Implementation

We have implemented the URRSA server and deployed on an internet facing machine: see Section 6 below. We now address some of the issues related to implementation. This architecture and flow of events during an authentication is illustrated in Figure 1. We use ASP.Net scripting to handle the the actions to be performed at the web server. There are three web services running on the server: `OTPRegistration`, `OTPLogin` and `OTPRefresh`, which we review in turn.

4.1 Registration Webservice

To use the service a user must first navigate to {`URRSA`}`/OTPRegistration` and get a list of one-time passwords. He enters the url and userID of an account he wishes to access and is assigned a randomly chosen set of keys. Recall that the url and userID pair uniquely identifies the user, and hence the set of keys issued. He also enters the password, *pwd*, for this account; and indicates the cellphone number he wishes the list sent to. The webservice then generates a list of 10 one-time passwords and sends them by SMS text message to the desired number.

For this step `OTPRegistration` interfaces to an SMS gateway service. There are several providers which expose programmable interface to make sending text messages simple. In our implementation we use Clickatell [7] which offers a variety of means of triggering the send message. In the simplest, after establishing an account and paying for credits, a message can be sent merely by navigating to: `http://api.clickatell.com/http/sendmsg?user=xxxxx&password=xxxxx&api_id=xxxxx&to=xxxxxxx&text=xxxxxx`, where the fields to be filled are the user account name, password, application id, destination phone number and SMS message. This could be invoked by causing the user's browser to navigate to the appropriate address once the OTP list has been calculated. While simple, this leaves our password to the SMS gateway (though not the user's password) in the clear on the trusted machine. While the machine is trusted by the user, the user is not necessarily trusted by the server, and this would potentially allow him to replay and exhaust the message budget at the gateway. Instead we use an API integrated with the server, which also causes the desired message to be sent. We are limited to only 10 passwords by the 160 character limit that applies to SMS messages. This step must be done at a trusted machine. The keys are stored at the server along with the url and userID, but no record of the password is kept.

In the event that the user does not have a cellphone he may elect to carry a copy on paper. In this case the webservice then generates a larger list of 30 one-time passwords which he prints and carries. The OTP's can be carried on a PDA, or an mp3 player that is capable of displaying text files. This has the advantage that storage is no longer an issue. As with any OTP system the user must protect the list (see coverage of attacks in Section 5).

4.2 Login Webservice

Now to login the user navigates to {`URRSA`}`/OTPLogin`. He is asked for the url and userID of the account he wishes to access. Since this pair uniquely identifies him this allows the proxy to retrieve the keys. For the k-th login the user is prompted to enter the k-th OTP from his list: OTPk. The server decrypts to get the true password. The user's browser is instructed to automatically open `https://`{`URRSA_1`}. Using the `onclick` event for the "Submit" button we can use, for example the Javascript `Open()` command, which causes a new window to open with a specified URL.

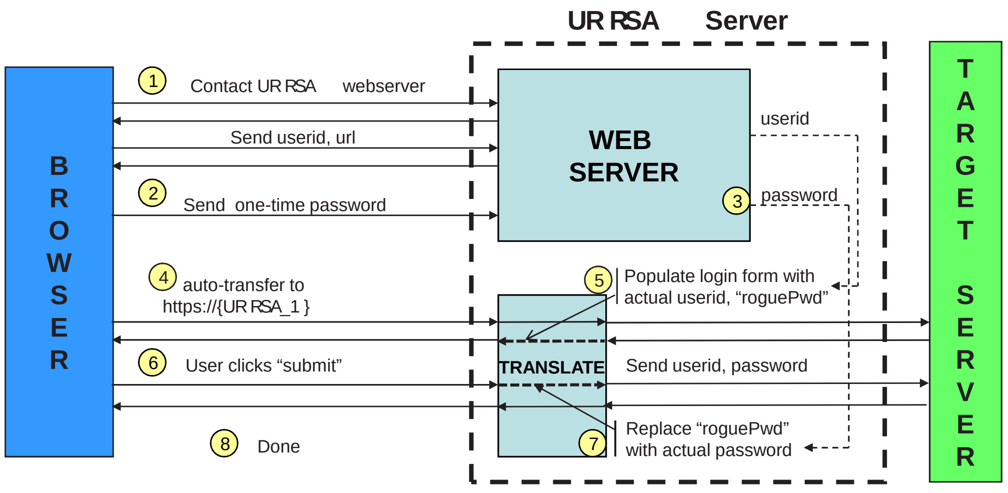

Fig. 1. The sequence of steps logging in using the URRSA service. See a description in Section A.2. The heart of the service is the translation which acts as a MITM service: it sits between the client browser and the login server and edits the request/response traffic between them. The implementation this part is described in detail in Section 4.2.

Backend processing To reliably translate as a MITM service between `https://{URRSA_1}` and `https://www.paypal.com` our URRSA implementation must perform [23]:

- request URL mapping
- request header mapping
- response header mapping
- response body link translations
- cookie re-assignment
- certificate replacement.

These changes can be implemented directly on the request/response stream using the `WebClient` and `HttpWebResponse` classes in .Net. There are also a variety of proxy applications that offer the ability to modify in real-time request/response traffic [8–10]. The most common mapping that must be performed on the request header is to the `Host` field. Requests that are constructed relative to a given host will require mapping of this field. For example, in requesting `https://www.BankOfAmerica.com/accounts` the GET request issued by the browser is for `/accounts` and the `Host` from which it is to be retrieved is in the header. However, when using a reverse proxy we want the browser to request `/accounts` from `{URRSA_1}`. The proxy must then map `{URRSA_1}` to `www.BankOfAmerica.com` as the request goes by.

Response headers often contain information about the document in the `Location` field. The most common case is to indicate the new location when a document has moved. This is a very common way to allowing sites to add and change web site content and point several access points at a single login page. Before forwarding the response back to the browser we have the proxy map `www.paypal.com` to `{URRSA_1}`. Cookies play an important rôle at many login servers. When a cookie is set by PayPal the same origin policy will cause the browser to return that cookie only with requests sent to PayPal. This is problematic, since the browser at the untrusted machine is connected to `{URRSA_1}` rather than `www.paypal.com`. We solve this by re-assigning the domain to which cookies belong in the response header.

When SSL connected to PayPal the browser receives a PayPal certificate signed by Verisign (a Certificate Authority (CA) trusted by most browsers). The URRSA server will maintain two SSL connections. From PayPal it receives a PayPal certificate signed by Verisign, just as a regular client would. For the SSL connection it maintains with the client it must have its own certificate, also signed by a CA trusted by the browser. Thus the user will never see a PayPal certificate, but rather one for {`URRSA_1`}.

The response content must have all references to the end server replaced with ones to the proxy. For example translate requests such as `https://www.paypal.com/images/logo.gif` to `https://`{`URRSA_1`}`/images/logo.gif`. Table 2 lists the rules to carry out the mappings referred to above.

The changes described so far allow us to translate for a single host. However, for many sites content is loaded from more than one host. For example, when logging into PayPal the browser loads content both from `www.paypal.com` and `www.paypalobjects.com`. For gmail content is loaded from `mail.google.com` and `www.google.com`, while for sites such as myspace as many as a dozen or more hosts can be involved. We solve this by having a single IP address, or host, at the proxy handle each host that is involved in a login at the end server. For example for Paypal the translations for `www.paypal.com` will be handled at {`URRSA_1`} while those for `www.paypalobjects.com` will be handled at {`URRSA_2`}. This has the major advantage that relative links in the response are resolved automatically without intervention by the proxy. This represents a major point of difference with reverse proxies based on CGIProxy: these use a single proxy host for any and all hosts involved in the login session. Thus all relative links must be found and translated. A detailed description of reverse proxying can be found in [26].

In addition, recall that we must insert the user's decrypted password into the request as the login page is submitted to the server. When a login page is loaded the response content (Step 5 of Figure 1) contains a password field. We auto-populate this field by replacing in the response body the string `type=‘‘password’’` with `type=‘‘password’’ value=‘‘roguePwd’’`. This causes the login page that appears to the user to have an already filled in password field (this is not of course the user's password, but rather the "roguePwd" string). We have a further rule that replaces in the request header the string "roguePwd" with the just decrypted user password (Step 6 of Figure 1).

Having the login page appear with an auto-populated password field serves a number of functions. First, it provides the sentinel value for which the URRSA server will search the request header to do the actual password switch. Second, many login pages will not allow submission with an empty password field, so the field must contain something. Finally, in having the field auto-filled with a value that is of no use to an attacker we greatly reduce the risk that the user reflexively types his password when faced with an empty login page.

Table 2 lists the rules to carry out the mappings between `www.paypal.com` and {`URRSA_1`} referred to above. A similar set of rules would translate between `www.paypalobjects.com` and {`URRSA_2`}. Essentially any server can be handled in this way, where we dedicate an IP address or host at the proxy to each host at the end server. A more scalable reverse proxy scheme is described in [26].

4.3 Refresh Webservice

When the user requires a new OTP he can re-register. Alternatively, he can have a new list generated and sent to his cell phone. This works as follows. A service {`URRSA`}`/OTPRefresh` resembles the {`URRSA`}`/OTPRegistration` service. The user identifies himself by giving the url and userID of the account for which he requires a new OTP list. However, instead of presenting his true password for encryption he submits the last OTP on his list. This allows the server to retrieve the key i. The proxy decrypts to get the true password, and then re-encrypts to form a new list and transmits them to the desired number. In this manner the user can repeatedly refresh his OTP list without having to return to a trusted machine. Of course, he must refresh the list before he uses the last OTP on his list. OTP Refresh is not available if the user carries the list by paper, since the proxy has no out-of-band channel to send a fresh list.

5 Attacks

5.1 Lost or Stolen OTP list

First, the technology is a one-time password (OTP) technique, and therefore, subject to the same kind of vulnerabilities as other OTP systems. It deserves emphasis that the list must be generated at a trusted location and should be kept carefully. The OTP list is sent by SMS text message to the user's cellphone, but printing and carrying a hardcopy is also supported. The list contains the url of the login server along with the OTP list, but not the userID. If the OTP sheet is lost or stolen the finder will possess a series of one-time passwords, but will not know the userID of the account for which they work. We recognize that this is an imperfect defence, and do not claim to have solved the problem of users who are careless or lose their OTP sheet (it is for this reason that we regard the phone as a better channel). However a user who discovers he has lost his OTP sheet can render it useless by generating a new sheet. If he can go to a trusted PC he generates a new sheet and the old one is worthless (since a new set of keys is generated). A user who cannot reach a trusted PC can still render the lost OTP sheet worthless by re-registering at an untrusted PC. By typing random characters instead of the true password he will receive one-time encryptions of junk, but this accomplishes a key reset at the service, rendering the lost OTP's useless. He cannot now use the service until reaching a trusted PC. A user who carries the OTP list both on a cell phone and by paper and who loses the paper can, of course, render the lost sheet useless by using the {`URRSA`}/OTPRefresh service.

5.2 Brute-force and Denial of Service

An attacker who wishes to guess or brute force the password will gain nothing by going through the service, since we do not protect strong secrets with weak ones. To login normally he would require the userID and password, to login via the service he requires the userID and the password encrypted with the correct key. Any lockout policies enforced by the login server (*e.g.* "Three strikes and you're out") remain in effect. An attacker who observes several logins or gains access to the entire list cannot brute-force the password.

The proxy itself is a likely point of attack. Observe, however, that the OTP Login Web interface (described in Section 4.2) is merely a conventional password web interface: it accepts text and password HTML form fields from the user and relays them to backend processing. Thus the web facing portion of the proxy is implemented with a tried and trusted password server. The fact that we use components with well-understood attack surfaces increases the expectation that attacking the system will be hard. Since at the backend credentials are stored only temporarily, a snapshot of the database would gain an attacker little. A rogue employee at the proxy would see at any given time only the credentials of users currently logging in.

Since we do not authenticate users it is possible to exhaust a user's OTP list by repeatedly invoking the OTPLogon service (with the correct url and userID but incorrect OTP's). This form of denial of service is possible, but gains the attacker nothing unless he can lure the user into typing the true password in the clear.

5.3 Session Hijacking and OTP Stealing

There are two main active attacks on the system: session hijacking and OTP stealing. Session hijacking is not addressed by our technique. Indeed, even long established OTP solutions such as SecurID [3] have this vulnerability. However session hijacking is a complicated attack that requires code tailored to each target login server. The fact that RSA has had considerable commercial success protecting high-value accounts in spite of this well-known vulnerability suggests that session-hijacking is not a common attack. In addition the techniques suggested in [21], which require explicit out-of-band authorization for important transactions, might present a way of addressing this problem.

OTP stealing is the technique that malware can employ to get OTP's from the user. There are a number of variations; the simplest is to allow a user to connect to {`URRSA`}`/OTPLogin`, have him enter the requested OTP and then fail to submit it. If the user assumes that he mis-typed he might try again, and give the attacker a second OTP. Depending on the user the attacker might gain anywhere between one and three OTP's this way (we assume that it is unlikely that a user will type more than that). This is a well-known attack on all static OTP systems; dynamic systems such as securID do not have this vulnerability since each OTP is good for only a few seconds. We do not eliminate this attack. However, we point out that an attacker who logs in with a stolen OTP has full access, but cannot change the account password. This is so, since almost all web services require the original password before allowing a change; while the attacker has one or more OTP's he cannot use these to derive the password. This restricts the attacker: if he has stolen three OTP's he can now login three times. He cannot change or get access to the true password and thus every access must continue to be done via URRSA. It is possible to restrict the types of actions that can be performed via the proxy. For example, submission of the HTML form that changes the user email, address or phone number might be forbidden. This can be accomplished by adding a rule to the translations in Table 2 that drops the request that contains any such POST. Finally, we point out that the OTP stealing attacker leaves a clue to his presence in that he must cause a login to fail for each OTP he steals (*i.e.* each OTP becomes worthless after its first use). Thus, in the absence of failed login attempts the user can be confident that no OTP has been stolen. A user who suspects that an OTP has been stolen can, of course, render them useless by connecting to {`URRSA`}`/OTPLogin`.

6 Status and Evaluation

We have implemented the system described and deployed on an internet-facing server. The service currently supports OTP logins to a number of sites. The entire server code is small enough that it can comfortably be hosted on a modest desktop machine. This makes self-hosting a real possibility: *i.e.* users who have a fixed IP address or domain name might run their own instance. This removes the necessity of trusting a proxy maintained by a third party.

The URRSA instance in its current form has been in active use by a small number of users for over six months. A large number of successful logins to have been handled. These were carried out from a variety of networks; *i.e.* machines that exist on home networks, are behind corporate firewalls, internet cafés and public library locations have all been tried. The service works well with Internet Explorer, Firefox, Safari and Opera. Users report that all of the functionality generally available at a server works well when accessed through the URRSA service. Users do not report perceptible delay, and the service is generally transparent to users once connection is established. The service is running at `www.urrsa.com`. It is not possible to invite general use as yet. However, for demonstration and verification purposes, we may allow restricted access as conditions permit.

7 Conclusions

We have described a system that allows users one-time password access to accounts without changing the server or the client. The method is entirely general and can be applied to almost any login server. Among the key contributions are a very simple user experience and a truly robust MITM translation. We do not authenticate users: thus there are no additional secrets to remember or tokens for the user to carry. The service acts as a transparent MITM between user and login server: thus there are no browser settings to be done or undone. We employ a simple mapping of the arbitrary input password to restricted character set OTP's: thus every OTP is readable without ambiguity no matter what display or font is used, can be transmitted over SMS, and can be entered even on unfamiliar keyboards without the use of meta keys.

Acknowledgements: the authors wish to thank Eric Lawrence, Ziqing Mao, Nikita Pandey, Erin Renshaw and Dany Rouhana for help with various stages of this work.

References

1. `http://labs.zarate.org/passwd/`.
2. `http://www.cl.cam.ac.uk/~mgk25/otpw.html`.
3. `http://www.rsasecurity.com`.
4. `http://www.kyps.net`.
5. `http://www.csoft.co.uk/sms/character_sets/encoding.htm`.
6. `http://www.jmarshall.com/tools/cgiproxy`.
7. `http://www.clickatell.com`.
8. `http://www.fiddlertool.com`.
9. `http://www.xk72.com/charles/`.
10. `http://www.portswigger.net/proxy`.
11. C. Herley and D. Florêncio. How To Login From an Internet Café without Worrying about Keyloggers. *Symp. on Usable Privacy and Security*, 2006.
12. W. Cheswick. Johnny Can Obfuscate: Beyond Mother's Maiden Name. *In Proc. Usenix HotSec*, 2006.
13. B. Coskun and C. Herley. Can "Something You Know" be Saved? *ISC 2008, Taipei.*
14. D. Florêncio and C. Herley. A Large-Scale Study of Web Password Habits. *WWW 2007, Banff.*
15. D. Florêncio and C. Herley. KLASSP: Entering Passwords on a Spyware Infected Machine. *ACSAC*, 2006.
16. D. Florêncio, C. Herley, and B. Coskun. Do Strong Web Passwords Accomplish Anything? *Proc. Usenix Hot Topics in Security*, 2007.
17. E. Gaber, P. Gibbons, Y. Matyas, and A. Mayer. How to make personalized web browsing simple, secure and anonymous. *Proc. Finan. Crypto '97.*
18. P. Golle and D. Wagner. Cryptanalysis of a Cognitive Authentication Scheme. *Symp. on Security and Privacy*, 2007.
19. N. Haller. The S/KEY One-Time Password System. *Proc. ISOC Symposium on Network and Distributed System Security*, 1994.
20. C. Herley and D. Florêncio. Phishing as a Tragedy of the Commons. *NSPW 2008, Lake Tahoe, CA.*
21. R. C. Jammalamadaka, T. W. van der Horst, S. Mehrotra, K. Seamons, and N. Venkasubramanian. Delegate: A Proxy based Architecture fort Secure Website Access from an Untrusted Machine. *Proc. ACSAC*, 2006.
22. L. Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 1981.
23. A. Luotonen. Web Proxy Servers. *Prentice Hall*, 1998.
24. M. Mannan and P.C. van Oorschot . Using a Personal Device to Strengthen Password Authentication from an Untrusted Computer. In *Financial Cryptography*, 2007.
25. M. Wu and S. Garfinkel and R. Miller . Secure Web Authentication with Mobile Phones. In *DIMACS Workshop on Usable Privacy and Security Software*, 2004.
26. Z. Mao and C. Herley. Robust Reverse Proxy Implementation. *MSR-TR.*

27. A. Pashalidis and C. J. Mitchell. Impostor: A single sign-on system for use from untrusted devices. *Proceedings of IEEE Globecom*, 2004.
28. T. Pering, M. Sundar, J. Light, and R. Want. Photographic Authentication through Untrusted Terminals. *IEEE Security and Privacy*, 2003.
29. B. Schneier. *Applied Cryptography.* Wiley, second edition, 1996.
30. T. C. Bell, J. G. Cleary and I. H. Witten. *Text Compression.* Prentice Hall, 1990.
31. D. Tan, P. Keryana, and M. Czerwinski. Spy-resistant keyboard: more secure password entry on public touch screen displays. *CHISIG'05.*
32. D. Weinshall. Cognitive Authentication Schemes Safe Against Spyware. *Symp. on Security and Privacy*, 2006.

A Additional Details

A.1 Pseudo-code of the mapping procedure

Here is a pseudo code illustrating the above encoding procedure:

```
//Encode
// get bits from input password P
BitString = 0;
TotalBits = 0;
for each character P[i] {
   PP = table128_lookup_character(P[i]);
   BitString = BitString <<7 + PP;
   TotalBits += 7;
}
//encrypt
Key = get next TotalBits bits from One Time Encryption Pad
BitString = BitString XOR Key;
//convert bits to OTPi characters
i = 0;
while TotalBits > 0 {
     PP = BitString AND 31
     OTPi[i] = table32[PP];
     BitString = BitString >> 5;
     TotalBits -= 5;
     i++;
}
```

A.2 Sequence of Events

1. User navigates to `http://{URRSA}/OTPLogon`, enters the userID and url (*e.g.* `www.paypal.com`). This allows the server to determine the key values, that were used to generate the OTP list.
2. For the k-th login the user enters the k-th one-time password OTPk
3. The server decrypts to get the true password *pwd*
4. Browser is auto-transferred to request the url via the URRSA service (*e.g.* we request `https://{URRSA_1}`).
5. Server requests `https://www.paypal.com`, receives the response and populates login form with userID and "roguePwd," and sends to user.
6. User receives pre-populated `https://www.paypal.com` page and clicks submit button.
7. Server receives request, replaces "roguePwd" with *pwd*, forwards to `https://www.paypal.com`.
8. Login proceeds and server continues in a MITM rôle until the user navigates from the site.

A.3 Login servers that do not use forms

Our system is applicable to login servers that use HTML forms and POST a password to be authenticated at the server. While this appears to account for the vast majority of login servers there are exceptions. Certain institutions implement an entirely proprietary authentication on their website using Flash or a comparable technology. For example FirstTech Credit Union uses only Flash on their login page `https://online.firsttechcu.com/`.

`http://{URRSA}/OTPRegistration`	Register login domain and userID and receive OTP list
`http://{URRSA}/OTPLogin`	Enter login domain, userID and requested OTP
`http://{URRSA}/OTPRefresh`	Enter login domain, userID and requested OTP to get new OTP list
`https://{URRSA_1}`	MITM service that performs mappings described in Section 4.2 and Table 2.

Table 1. A summary of the services described. The variable `URRSA` represents the host domain name or IP address (*e.g.* we give the hostname of our implementation in Section 6). The last service is requested after an OTP has been received from the user and decrypted.

Modify	Search for	Replace with
Request Header	{`URRSA_1`}	`www.paypal.com`
Response Header	`www.paypal.com`	{`URRSA_1`}
Response Header	domain=`.paypal.com`	domain={`URRSA_1`}
Response Body	`www.paypal.com`	{`URRSA_1`}
Response Body	`www.paypalobjects.com`	{`URRSA_2`}
Response Body	type="password"	type="password" value="roguePwd"
Request Header	roguePwd	Actual password as decrypted

Table 2. The translation rules applied to the request/response stream to reverse proxy between the two hosts `www.paypal.com` and {`URRSA_1`}.