Dos and Don’ts of Client Authentication on the Web

Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster
{fubob, sit, kendras, feamster}@mit.edu
MIT Laboratory for Computer Science
http://cookies.lcs.mit.edu/

Abstract

Client authentication has been a continuous source of problems on the Web. Although many well-studied techniques exist for authentication, Web sites continue to use extremely weak authentication schemes, especially in non-enterprise environments such as store fronts. These weaknesses often result from careless use of authenticators within Web cookies. Of the twenty-seven sites we investigated, we weakened the client authentication on two systems, gained unauthorized access on eight, and extracted the secret key used to mint authenticators from one.

We provide a description of the limitations, requirements, and security models specific to Web client authentication. This includes the introduction of the *interrogative adversary*, a surprisingly powerful adversary that can adaptively query a Web site.

We propose a set of hints for designing a secure client authentication scheme. Using these hints, we present the design and analysis of a simple authentication scheme secure against forgeries by the interrogative adversary. In conjunction with SSL, our scheme is secure against forgeries by the active adversary.

1 Introduction

Client authentication is a common requirement for modern Web sites as more and more personalized and access-controlled services move online. Unfortunately, many sites use authentication schemes that are extremely weak and vulnerable to attack. These problems are most often due to careless use of authenticators stored on the client. We observed this in an informal survey of authentication mechanisms used by various popular Web sites. Of the twenty-seven sites we investigated, we weakened the client authentication of two systems, gained unauthorized access on eight, and extracted the secret key used to mint authenticators from one.

MIT Technical Report 818. Appendix updated on September 7, 2001. This research was supported by a USENIX scholars fellowship. A shorter version of this document was originally published in the Proceedings of the 10th USENIX Security Symposium, Washington, D.C., August, 2001.

This is perhaps surprising given the existing client authentication mechanisms within HTTP [16] and SSL/TLS [11], two well-studied mechanisms for providing authentication secure against a range of adversaries. However, there are many reasons that these mechanisms are not suitable for use on the Web at large. Lack of a central infrastructure such as a public-key infrastructure or a uniform Kerberos [40] contributes to the proliferation of weak schemes. We also found that many Web sites would design their own authentication mechanism to provide a better user experience. Unfortunately, designers and implementers often do not have a background in security and, as a result, do not have a good understanding of the tools at their disposal. Because of this lack of control over user interfaces and unavailability of a client authentication infrastructure, Web sites continue to reinvent weak home-brew client authentication schemes.

Our goal is to provide designers and implementers with a clear framework within which to think about and build secure Web client authentication systems. A key contribution of this paper is to realize that the Web is particularly vulnerable to adaptive chosen message attacks. We call an adversary capable of performing these attacks an *interrogative adversary*. It turns out that for most systems, every user is potentially an interrogative adversary. Despite having no special access to the network (in comparison to the eavesdropping and active adversary), an interrogative adversary is able to significantly compromise systems by adaptively querying a Web server. We believe that, at a minimum, Web client authentication systems should defend against this adversary. However, with this minimum security, sites may continue to be vulnerable to attacks such as eavesdropping, server impersonation, and stream tampering. Currently, the best defense against such attacks is to use SSL with some form of client authentication; see Rescorla [36] for more information on the security and proper uses of SSL.

In Section 2, we describe the limitations, requirements, and security models to consider in designing Web client authentication. Using these descriptions, we codify the principles underlying the strengths and weaknesses of existing systems as a set of hints in Section 3. As an example, we design a simple and flexible authentication scheme in Section 4. We implemented this scheme and analyzed its security and performance; we present these findings in Sections 5 and 6. Section 7 compares the work in this paper to prior and related work. We conclude with a summary of our results in Section 8. The appendix describes several existing client authentications schemes in detail.

2 Security models and definitions

Clients want to ensure that only authorized people can access and modify personal information that they share with Web sites. Similarly, Web sites want to ensure that only authorized users have access to the services and content it provides. Client authentication addresses the needs of both parties.

Client authentication involves proving the identity of a *client* (or user) to a *server* on the Web. We will use the term "authentication" to refer to this problem. Server authentication, the task of authenticating the server to the client, is also important but is not the focus this paper.

In this section, we present an overview of the security models and definitions relevant in client authentication. We begin by describing the practical limitations of a Web authentication system. This is followed by a discussion of common requirements. We then characterize types of breaks and adversaries.

2.1 Practical limitations

Web authentication is primarily a practical problem of deployability, user acceptability, and performance.

Deployability

Web authentication protocols differ from traditional authentication protocols in part because of the limited interface offered by the Web. The goal is to develop an authentication system by using the protocols and technologies commonly available in today's Web browsers and servers. Authentication schemes for the Internet at large cannot rely on technology not widely deployed. For example, Internet kiosks today do not have smart card readers. Similarly, home consumers currently have little incentive to purchase smart card readers or other hardware token systems.

The client generally speaks to the server using the Hypertext Transfer Protocol (HTTP [14]). This may be spoken over any transport mechanism but is typically either TCP or SSL. Since HTTP is a stateless, sessionless protocol, the client must provide an *authentication token* or *authenticator* with each request.

Computation allows the browser to transform inputs before sending them to the server. This computation may be in a strictly defined manner, such as in HTTP Digest authentication [16] and SSL, or it may be much more flexible. Flexible computation is available via Javascript, Java, Tcl, ActiveX, Flash, and Shockwave. Depending on the application, these technologies could perhaps assist in the authentication process. However, most of these technologies have high startup overhead and mediocre performance. As a result, users may choose to disable these technologies. Also, these extensions may not be available on all operating systems and architectures. Any standard authentication scheme should be as portable and lightweight as possible, and therefore require few or no browser extensions. Thus for today's use, any authentication scheme should avoid using client computation for deployability reasons. If absolutely necessary, Javascript and Java are commonly supported.

Client state allows the client's browser to store and reuse authenticators. However, storage space may be very limited. In the most limited case, the browser can only store passwords associated with realms (as in HTTP Basic authentication [16]). A more flexible form of storage which is commonly available in browsers is the cookie [24, 31]. Cookies allow a server store a value on a client. In subsequent HTTP requests, the client automatically includes the cookie value. A number of attributes can control how long cookies are kept and to which servers they are sent. In particular, the server may request that the client discard the cookie immediately or keep it until a specified time. The server may also request that the client only return the cookie to certain hosts, domains, ports, URLs, or only over secure transports. Cookies are the most widely deployed mechanism for maintaining client state.

User acceptability

Web sites must also consider user acceptability. Because sites want to attract many users, the client authentication must be as non-confrontational as possible. Users will be discouraged by schemes requiring work such as installing a plug-in or clicking away dialog boxes.

Performance

Stronger security protocols generally cost more in performance. Service providers naturally want to respond to as many requests as possible. Cryptographic solutions will usually degrade server performance. Authentication should not needlessly consume valuable server resources such as memory and clock cycles. With current technology, SSL becomes unattractive because of the computational cost of its initial handshaking.

2.2 Server security requirements

The goals of a server's authentication system depend on the strength and granularity of authentication desired. Granularity refers to the fact that some servers identify individual users throughout a session, while others identify users only during the first request. A fine-grained system is useful if specific authorization or accountability of a user is required. A coarse-grained system may be preferred in situations where partial user anonymity is desired.

A simple example of a coarse-grained service is a *subscription service* [41]. Subscription services merely wish to verify that a user has paid for the service before allowing access to read-only content. During the initial request, a user could authenticate with a username and password. Unless the service allows customization, subsequent requests need only verify that a user has been authenticated without knowing the user's actual identity. A trusted third party could handle the initial authentication of a user. Some specific examples of sites that only require this level of authentication are newspapers (e.g., `WSJ.com`), online libraries (e.g., `acm.org`), and adult entertainment (e.g., `playboy.com`).

However, most sites customize the data sent back to users. This naturally requires a fine-grained system. Each user must be identified specifically to use their preferences. Examples of this include sites that allow users to customize look-and-feel (e.g., `slashdot.org`), sites that filter information on behalf of the user (e.g., `infobeat.com`), or sites which provide online identities (e.g., `hotmail.com`).

2.3 Confidentiality and privacy

Confidentiality is not strictly related to authentication but it is worth mentioning as well, since it can be provided by cryptography and since it is often confused with authentication. A system that provides confidentiality protects traffic from disclosure by anyone except the sender and recipient. In contrast, a system that provides authentication ensures that the person sending or receiving the data is indeed who they claim to be. This may be confusing because SSL, the only widely deployed mechanism for providing confidentiality of HTTP transactions, provides options for both authentication and confidentiality. The distinction between confidentiality and authentication is further blurred by the practice of current browsers of displaying a single padlock whose meaning is ambiguous.

Typically, servers choose to provide confidentiality for only certain special data by using SSL. For example, financial data require confidentiality. Sites that deal with such information, *online brokerages*, may be auction sites (e.g., `ebay.com`), banks and other financial service providers (e.g., `etrade.com`), or online merchants (e.g., `FatBrain.com`).

Another issue commonly associated with authentication is user *privacy*. Privacy refers to protecting the data available on the server from access by unauthorized parties. While often the information provided by the server is not itself secret, one does not usually want unknown parties discovering their personal interests. For example, a user may sign up to see discount airfares to San Francisco or select stocks in a portfolio for updated stock quotes. While the fact that US Airways is offering a low fare or that Cisco stock has shed four points is not in any way secret, it may be telling to find out if a particular user is interested in that information. Therefore servers often need to provide ways to keep personalized data private. Privacy can be achieved by using secure authentication and providing confidentiality.

2.4 Breaks

An adversary's goal is to break an authentication scheme faster than by brute force. Here we use terminology loosely borrowed from cryptography to characterize the kinds of breaks an adversary can achieve [18, 30].

In an *existential forgery*, the adversary can forge an authenticator for at least one user. However, the adversary cannot choose the user. This may be most interesting in the case where authenticators protect access to a subscription service. While an existential forgery would not give an adversary access to a chosen user's account, it would allow the adversary to access content without paying for it. This is the least harmful kind of forgery.

In a *selective forgery*, the adversary can forge an authenticator for a particular user. This adversary can access any chosen user's personalized content, be it Web e-mail or bank statements.

Note that a forgery implies the construction of a new authenticator, not one previously seen. In a traditional *replay attack*, the adversary is merely reusing a captured authenticator.

Finally, a *total break* results in recovery of the secret key used to mint authenticators. This is the most serious break in that it allows the adversary to construct valid authenticators at any time for all users.

2.5 Adversaries

We consider three kinds of adversaries that attack Web authentication protocols: the *interrogative adversary*, the *eavesdropping adversary*, and the *active adversary*. Each successive adversary possesses all the abilities of the weaker adversaries. Note that our definitions differ somewhat from tradition. Our adversaries gather information and apply this information to achieve a break. The adversaries differ from each other only in their information gathering ability.

Interrogative adversary

The *interrogative adversary* can make a reasonable number of queries of a Web server. It can adaptively choose its next query based on the answer to a previous query. We named this the interrogative adversary because the adversary makes many queries, but lacks the ability to sniff the network.

The ability to make queries is surprisingly powerful. The adversary can pass attempted forgeries to the server's verification routine. By creating new accounts on a server, the adversary can obtain the authenticator for many different usernames. This is possible on any server that allows account creation without some form of out-of-band authentication (e.g., credit cards) to throttle requests. In this paper we assume no such throttle exists.

The interrogative adversary can also use information publicly available on the server. A server may publish the usernames of valid account holders, perhaps in a public discussion forum. An adversary attacking this server might find this list useful.

In more theoretical terms, the interrogative adversary may treat the server as an oracle. An interrogative adversary can carry out an *adaptive chosen message attack* by repeatedly asking for the server to mint or verify authenticators [18].

Eavesdropping adversary

The *eavesdropping adversary* can see all traffic between users and the server, but cannot modify any packets flowing across the network. That is, the adversary can sniff the network and replay authenticators. This adversary also has all the abilities of the interrogative adversary.

An eavesdropping adversary can apply its sniffed information to attempt a break. Computer systems research would consider this an active attack; we do not. This style of definition is more common in the theory community where attacks consist of an information gathering process, a challenge, another optional information gathering process, and then an attempted break [3].

Active adversary

The *active adversary* can in addition see and modify all traffic between the user and the server. This adversary can mount man-in-the-middle attacks. In the real world, this situation might arise if the adversary controls a proxy service between the user and server.

3 Hints for Web client authentication

We present several hints for designing, implementing, and selecting a scheme for client authentication on the Web. Some of these hints come from our experiences in breaking authentication schemes in use on commercial Web sites. Others come from general knowledge or security discussion forums [45]. Following these hints is neither necessary nor sufficient for security. However, they would have prevented us from breaking the authentication schemes on several Web sites mentioned in this section. Most of these sites have subsequently repaired the problems we identified. These incidents help to demonstrate the usefulness of these hints. The details of our analysis are documented in the appendix.

Although we give advice on how to perform client authentication on the Web, we certainly do not advocate having everyone design their own security systems. Rather, we hope that these hints will assist researchers and developers of Web client authentication and dissuade persons unfamiliar with security from implementing home-brew solutions.

The hints fall into three categories. Section 3.1 discusses the appropriate use of cryptography. Section 3.2 explains why passwords must be protected. Section 3.3 offers suggestions on how to protect authenticators.

3.1 Use cryptography appropriately

Use of cryptography is critical to providing authentication. Without the use of cryptography, it is not possible to protect a system from the weakest of adversaries. However, designing cryptographic systems is a difficult and subtle task. We offer some hints to help guide the prospective designer in using the cryptographic tools available.

Use the appropriate amount of security

An important general design hint is to Keep It Simple, Stupid [26]. The more complex the scheme, the harder it is to develop compelling arguments that it is secure. If you are designing or selecting a system, choose one that provides the right amount of security for your needs. For example, an online newspaper cares about receiving compensation for content. An online brokerage cares about confidentiality, integrity, and authentication of information. These security needs are very different and can be satisfied by different systems. There are usually tradeoffs between the user interface, usability, and performance. Choosing an overly complex or featureful system will make management more difficult; this can easily result in security breaches.

Do not be inventive

It is a general rule in cryptography that secure systems should be designed by people with experience. Time has repeatedly shown that systems designed or implemented by amateurs are weak and easily broken. Thus, while we encourage research in developing authentication systems for the Web, it is very risky to design your own authentication system. This is closely related to our next hint. If you do choose to implement your own scheme, you should make your protocol publicly available for review.

Do not rely on the secrecy of a protocol

A security system should not rely on the secrecy of its protocol. A protocol whose security relies on obscurity is vulnerable to an exposure of the protocol. If there are any flaws, such an exposure may reveal them. For example, a secret system can be probed by an interrogative adversary to determine its behavior to valid and invalid inputs. This technique allowed us to reverse engineer the `WSJ.com` client authentication protocol. By creating several valid accounts and comparing the authenticators returned by the system, we were able to determine that the authenticator was the output of `crypt` (salt, username + secret string) where + denotes concatenation. Once we understood the format of the authenticator, we were able to quickly recover the secret string, "March20", by mounting an adaptive chosen message attack. The program, included in the appendix, runs in 128×8 queries rather than the intended 128^8. Assuming each query takes 1 second, this program finishes in 17 minutes instead of the intended 2×10^9 years. This information constitutes a total break, allowing us to mint valid authenticators for all users.

On the other hand, Open Market published their design and implementation [28].

Instead of relying on the secrecy of the scheme, rely on the secrecy of a well-selected set of keys. Ensure that the protocol is public so that it can be reviewed for flaws and improved. This will lead to a more secure system than a private protocol which appears undefeatable but may in practice be fairly easy to break. If you are hesitant to reveal the details of an authentication scheme, then you likely fear that it is vulnerable to an attack by an interrogative adversary.

Understand the properties of cryptographic tools

When designing an authentication scheme, cryptographic tools are critical. These include hash functions such as SHA-1 [15], authentication codes like HMAC [23], and higher-level protocols like SSL [11]. The properties each tool must be understood.

For example, SSL alone does not provide user authentication. Although SSL can authenticate users with X.509 client certificates, commercial Web sites rarely use this feature because of PKI deployment problems. Instead, SSL is used to provide confidentiality of authentication tokens and data. However, confidentiality does not ensure authentication.

Misunderstanding the properties of SSL made `FatBrain.com` vulnerable to selective forgeries by an interrogative adversary. In an earlier scheme, their authenticator consisted of a username and a session identifier based on a global sequence number. Since this number was global, an interrogative adversary could guess the session identifier for a chosen victim and make an SSL request with this session identifier. Here, the use of SSL did not make the system secure.

A more detailed example comes from a misuse of a hash function. One commonly (and often incorrectly) used input-truncating hash function is the Unix `crypt()` function. It takes a string input and a two-character salt to create a thirteen-character hash [30]; it is believed to be almost as strong as the underlying cryptographic cipher, DES [43]. However, `crypt()` only considers the first eight characters of its string input. This truncation property must be taken into account when using it as a hash.

The original `WSJ.com` authentication system failed to do so, which made our break possible. Since the input to `crypt()` was the username concatenated with the server secret, the truncation property of `crypt()` meant that the secret would not be hashed if the username was at least eight characters long. This means authenticators for long usernames can be easily created, merely with knowledge of the username. Additionally, the algorithm will produce an identical authenticator for all usernames that match in the first eight characters. This can be seen in Figure 1.

It is likely that `WSJ.com` expected this construction to act like a secure message authentication code (MAC). A message authentication code is a one-way function of both its input and a secret key that can be used to verify the integrity of the data [42]. The output of the function is deterministic and relatively short (usually sixteen to twenty bytes). This means that it can be recalculated to verify that the data has not been tampered with.

However, the `WSJ.com` authenticator was just a deterministic value which could always be computed from the first eight characters of the username and a fixed secret. While HTTP Basic authentication [16] (which uses no cryptography at all) is secure against an existential forgery of an interrogative adversary, the original `WSJ.com` scheme fell to a total break by the interrogative adversary.

Thus, when possible you should use a secure message authentication code. Certain cryptographic constructions have subtle weaknesses [30], so you should take great care in choosing which algorithm to employ. We recommend the use of HMAC-SHA1 [23]. This algorithm prevents many attacks known to defeat simple constructions. However, as we will see in Section 6, use of secure message authentication code is more expensive than an input-truncating hash such as `crypt()`.

Do not compose security schemes

It is difficult to determine the effects of composing two different security systems. Breaking one may allow an adversary to break the other. Worse, simply composing the schemes may have adverse cryptographic side effects, even if the schemes are secure in isolation. Menezes et al explain in remark 10.40 how using a single key pair for multiple purposes can compromise security [30]. The use of a single key for authentication and confidentiality leads to compromise of both if that key is stolen. On the other hand, if separate keys are used, a break of the authentication will not affect the confidentiality of past messages and vice versa.

`FatBrain.com` had two separate user authentication systems. To purchase a book, a user entered a username and password at the time of purchase. Future purchases required reauthentication. The account management Web pages had a separate security scheme which was stateful. After the user entered a username and password, FatBrain established a session identifier in the URL path. In this way, users could navigate to other parts of the account management system without having to tediously re-enter the password. Unfortunately, the security hole discussed in Section 3.3 allowed an adversary to gain access to the account management system for an arbitrary user by guessing a valid session identifier. The account management system includes an option to change a user's registered email address. By changing the email address of a victim's account and then selecting "mail me my password," an adversary could break into to the book purchasing part of the system, despite the fact that it was secure in isolation.

3.2 Protect passwords

Passwords are the primary means of authenticating users on the Web today. It is important that any Web site guard the passwords of its users carefully. This is especially important since users, when faced with many Web sites requiring passwords, tend to reuse passwords across sites.

Limit exposure of passwords

Compromise of a password completely compromises a user. A site should never reveal a password to a user. For instance, `ihateshopping.net` included the user's password as a hidden form variable. A valid user should already know the password; sending it unnecessarily over the network gives the eavesdropping adversary more opportunity to sniff the password. Furthermore, sites should use the "password" field type in HTML forms. This hides the password as it is typed in and prevents an adversary from peeking over a user's shoulder to copy the password.

Even for non-secure Web sites, users should have the option to authenticate over SSL. That is, users should not type passwords over HTTP. Passwords sent over HTTP are visible to eavesdropping adversaries sniffing the network and active adversaries impersonating servers. Because users often have the same password on multiple

username	`crypt()` output	authentication cookie
bitdiddle	MaRdw2J1h6Lfc	bitdiddleMaRdw2J1h6Lfc
bitdiddler	MaRdw2J1h6Lfc	bitdiddlerMaRdw2J1h6Lfc

Figure 1: Comparison of `crypt()` and `WSJ.com` authentication cookies. The last field represents the username prepended to the output of the `crypt()` function. The input to the `crypt()` function is the username prepended to the string "March20".

servers, a stolen password can be extremely damaging. To protect against such attacks, a server could require users to conduct the login over an SSL connection to provide confidentiality for the password exchange; upon successful completion of the login exchange, the server can then set a cookie with an unforgeable authenticator for use over HTTP. The authenticator can be designed to limit the spread of damage, whereas passwords can not.

Prohibit guessable passwords

Many Web sites advise users to choose memorable passwords such as birthdays, names of friends or family, or social security numbers. This is extremely poor advice, as such passwords are easily guessed by an attacker who knows the user. Even without bad advice, passwords are fairly guessable [32]. Thus, servers ought to prohibit users from using any password found in a dictionary; such passwords are vulnerable to dictionary attacks. Servers can reduce the effectiveness of on-line dictionary attacks by restricting the number of failed login attempts or requiring a short time delay between login attempts.

Unfortunately, implementing this requirement will make a Web site less appealing to use since it makes passwords harder to remember.

Reauthenticate before changing passwords

In security-sensitive operations such as password changing, a server should require a user to reauthenticate. Otherwise, it may be possible for an adversary to replay an authentication token and force a password change, without actual knowledge of the current password.

3.3 Handle authenticators carefully

Authenticators are the workhorse of any authentication scheme. These are the tokens presented by the client to gain access to the system. As discussed above, authenticators protect passwords by being a short-term secret; the authenticator can be changed at any time whereas passwords are much less convenient to change.

Make authenticators unforgeable

Many sites have authenticators that are easily predictable. For instance, we noticed that `highschoolalumni.com` uses ID numbers and email addresses inside cookies to authenticate users. An interrogative adversary can find this information in the publicly available alumni database and mint an authenticator for an any user.

Authenticators often contain keys that function as session identifiers. These identifiers should be cryptographically random; statistical randomness is not sufficient. The Allaire Cold Fusion Web server issues CFTOKEN session identifiers which come from a linear congruential number generator [2]. As described above, `FatBrain.com` used essentially a global sequence number. While these numbers may be appropriate for tracking users, it is possible for an adversary to deduce the next output, and hence the next valid session identifier. This may allow the adversary access the information of another user.

Authenticators may also contain other information that the system will accept to be true. Thus, they must also be protected from tampering. This is done by use of a message authentication code (MAC). Because message authentication codes require a secret key, only an entity with knowledge of the key can recreate a valid code. This makes the codes unforgeable since no adversary should possess the secret key. Use only strong cryptographic hash functions. Do not use CRC codes or other non-cryptographic hashes, as such functions are often trivial to break.

Relatedly, when combining multiple pieces of data to input into a message authentication code, be sure to unambiguously separate the components. Otherwise, a cryptographic splicing attack may defeat the message authentication code. Since most inputs are text, this can be done using some character that is known not to appear in the input fragments. If components are not clearly separated, multiple inputs can lead to the same outputs. For example, "usernameaccess" could come from "username" followed by "access" or "user" followed by "nameaccess"; better to write "username&access" to ensure that the interpretation is unambiguous. Of course, care must be taken to prevent the username from containing an ampersand!

Protect authenticators that must be secret

Some systems believe that they are secure against eavesdropping adversaries because they send their authenticators over SSL. However, a secure transport is ineffective if the authenticators leak through plaintext channels. We describe two ways that authenticators are sent over SSL and mistakes which can lead to the authenticator leaking into plaintext.

One method is to set the authenticator as a cookie. When doing so, it is usually appropriate to set the Secure flag on cookies sent over SSL. When set to true, this flag instructs a Web browser to send the cookie over SSL only. A number of SSL Web sites neglect to set this flag. This simple error can completely nullify the benefits of SSL. For instance, customers of SprintPCS can view their account information and make equipment purchases online. To authenticate, a user enters a phone number and password over SSL. SprintPCS then sets a cookie which acts as an authenticator. Anyone with the cookie can log in as that user. The protocol so far is reasonably secure. However, because SprintPCS does not set the Secure flag on their authentication cookie, the authenticator travels in plaintext over HTTP whenever a user visits the main SprintPCS Web page. We believe that SprintPCS intended to protect against eavesdropping adversaries. Nevertheless, an eavesdropping adversary can access a victim's account with a replay because the cookie authenticator leaks over HTTP.

A second method of setting an authenticator is to include it as part of the URL. Though the HTTP 1.1 specification [14] recommends against this, it easy to do and sites still use this. The problem with this method is that it too can leak authenticators through plaintext channels. If a user follows a link from one page to another, the Web browser usually sends the Referer [sic] header. This field includes the URL of the page from which the current request originated. As described in Section 14.36 of the HTTP specification, the Referer field is normally used to allow a server to trace back-links for logging, caching, or maintenance purposes. However, if the URL of the linking page includes the authenticator, the server will receive a copy of the authenticator in the HTTP header. Section 15.1.3 of the specification recommends that clients should not include a Referer header in a non-secure HTTP request if the referring page was transferred with a secure protocol for exactly this reason. However, this is not a requirement; browsers such as Netscape and Lynx send the Referer header without any warning.

This can be exploited via a cross-site scripting attack [9]. An adversary can cause a user to execute arbitrary code and offer the user a link from a secure URL including the authenticator (that appears legitimate) to a link of the adversary's choosing. If the user selects the link, the Referer field in the request may include the authenticator, making it available to an eavesdropping adversary. Worse, the link could point to the adversary's machine. Then no eavesdropping is necessary to capture the authenticator. If the attacker is clever and uses an SSL server to host the attack, most browsers will not indicate that anything untoward is happening since they only warn users about transitions from SSL to non-SSL links.

Therefore, be careful when setting authenticators in cookies and follow the recommendation of the HTTP 1.1 specification by not using authenticators in URLs.

Avoid using persistent cookies

A *persistent* cookie is written to a file on the user's system; an *ephemeral* or *temporary* cookie is only stored in the browser's memory and disappears when the user exits the browser. An error in the way the browser or user handles the cookie file may make it accessible over the Internet, exposing the user's cookies to anyone who knows where to look. For instance, certain queries to search engines can produce many cookie files accidentally placed on the Web. This is documented in the appendix. If a persistent cookie in a leaked file contains an authenticator, an adversary can simply copy the cookie and break into the user's account. In addition, if the user accesses the account from a public system (say at a library or Internet café) and receives a persistent authentication cookie on that system, any subsequent user of that system can access the account. For these reasons, persistent cookies should not be considered private. Do not store authenticators in persistent cookies.

Limit the lifetime of authenticators

A good design must also gracefully handle the compromise of tokens which are designed to be secret. To limit the amount of damage a leaked authenticator can cause, limit its lifetime.

For authenticators that are stored in user cookies, do not rely on the cookie expiration field for secure expiration. Since the client is responsible for enforcing that expiration, a malicious client can set the lifetime arbitrarily.

Netscape users can manually extend these expirations by simply editing a text file. We were able to indefinitely extend the lifetime of our `WSJ.com` cookie authenticator even though `WSJ.com` set the cookie to expire in 11 hours. This was not extremely alarming, but if an adversary stole a cookie (as described in Section 3.3), there would be no way to revoke the adversary's access. The problem was compounded because the cookie authenticator remained the same even if a user's password changed. This prevented the `WSJ.com` site from easily revoking access to a compromised account.

To prevent unauthorized cookie lifetime extensions, include a cryptographically unalterable timestamp in the value of the cookie, or store the expiration time in a user-inaccessible place on the server. Securely binding expirations to authenticators limits the damage caused by a stolen authenticator.

Note that an authenticator that is stored in a cookie can be replayed, regardless of its expiration time, if it is leaked. By definition, unless the client uses computation, it can only send unmodified cookies back to the server. If replay prevention is desired, the authenticator must be kept confidential and changed after each use. In that case, it might be necessary to record recently received authenticators and verify that newly received authenticators are not replays.

Bind authenticators to addresses

It can also be useful to tie authenticators to specific network addresses. This helps protect against replay attacks by making it more difficult for the adversary to successfully reuse the authenticator. In addition to acquiring the authenticator, the adversary must appear to originate from the same network address for which the authenticator was minted. However, this may prematurely invalidate authenticators issued to mobile DHCP users.

4 Design

In this section we present a scheme for performing client authentication. This design is intended to be an example of a simple system that follows the hints provided in Section 3. We do not claim that the scheme is novel, but we do claim that the concepts and design process are not extensively discussed in literature. We present a brief security analysis of the schemes in Section 5.

Our scheme provides a personalizable authenticator which allows the server to statelessly verify the authenticity of the request and its contents. The server can explicitly control the valid lifetime of the authenticator as well. The authenticator can include all the information needed to service a request, or can be used as a key to refer to session information stored on the server.

The overall operation of this scheme is shown in Figure 2. We assume that the user has an existing account on the server which is accessed via a username and password. At the start of each session, the server receives the username and password, verifies them, and sets an authentication cookie on the user's machine. Since cookies are widely supported, this makes the system portable. Subsequent requests to the server include this cookie and allow the server to authenticate the request. The design of each cookie ensures that a valid cookie can only be created by the server; therefore anyone possessing a valid cookie is authorized to access the requested content on the server.

Our scheme is designed to be secure against an interrogative adversary, as we believe that most of the schemes we evaluated were designed with this type of adversary in mind. However, because SSL with server authentication provides confidentiality and integrity, layering our design on top of SSL can provide an authentication system secure against an active adversary.

4.1 Cookie Recipe

The recipe for our cookie follows easily from the hints presented in Section 3. We create an unforgeable authenticator that includes an explicit expiration time. We use HTTP state (i.e. cookies) to store this authenticator with the client. The value of this cookie is shown here:

$$\texttt{exp=}t\texttt{\&data=}s\texttt{\&digest=MAC}_k\texttt{(exp=}t\texttt{\&data=}s\texttt{)}$$

The expiration time is denoted t and is expressed as seconds past 1970 GMT. The data string s is an optional parameter denoting arbitrary data that the server wishes to associate with the client. Finally, the cookie includes a MAC for the cleartext expiration and data.

Our cookie requires the use of a *non-malleable* MAC; that is, one where it is intractable to generate a valid ciphertext from a plaintext message related to a plaintext message with a known ciphertext [12, 23]. That is, no adversary can generate a valid ciphertext without both the server's secret key and the plaintext, no matter how many samples of valid plaintext/ciphertext pairs the adversary has. Examples of keyed, non-malleable MACs are HMAC-MD5 and HMAC-SHA1 [23].

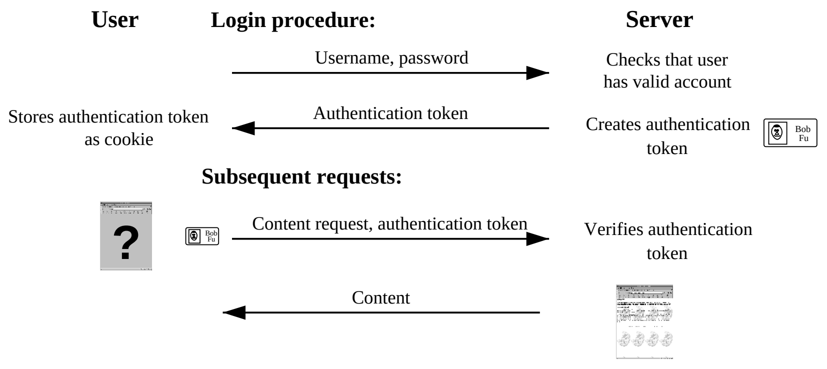
User
Login procedure:
Server
Username, password
Checks that user has valid account
Stores authentication token as cookie
Authentication token
Creates authentication token
Bob Fu
Subsequent requests:
?
Bob Fu
Content request, authentication token
Verifies authentication token
Content

Figure 2: One-exchange authentication system.

4.2 Discussion

Selecting an expiration time t is a trade-off between limiting the damage that can be done with a leaked authenticator and requiring the user to reauthenticate. Yahoo!, for example, allows users to specify what expiration interval they prefer for authenticators that control access to sensitive data [46]. This allows the user to control the trade-off. On the other hand, for insensitive data, it makes sense for the server to make the choice. For example, a newspaper might want cookies to be valid for only a day, whereas a magazine might allow sessions to be valid for a month (as if the user were buying a single issue).

The value s may be any information specific to the user that the server wishes to access without maintaining server state. This may be anything from a session identifier to a username. Beware that this data is not encrypted so sensitive information should not be stored here; if sensitive data is needed, we recommend that a cryptographically random session identifier be used. This will prevent information leaks from compromising a user's privacy. On the other hand, if sensitive user information is required to handle only a small percentage of the content requests, the authenticator can contain the information needed to service the majority of requests. This way the server can avoid doing a possibly expensive look-up with every request.

A server may also choose to leave s empty (and removing the `data` parameter from the cookie). This might be useful in the case where authentication must expire, but all users are essentially the same. A plausible example of this might be a pay-per use service, such as a newspaper.

4.3 Authentication and revocation

To authenticate a user, the server retrieves the cookie and extracts the expiration. If the cookie has not expired, the server recalculates the MAC in the `digest` parameter of the cookie. Since the server is the only entity who knows the key k, the properties of the `MAC` function imply that a valid cookie was generated by the server. So long as the server only generates cookies for authenticated users, any client with a valid cookie is a valid user.

This scheme does not provide a mechanism for secure *revocation*; that is, ending the user's session before the expiration time is up. The easiest option is for the server can instruct the client to discard the authentication cookie. This will usually be adequate for most applications. However, a client who has saved the value of the cookie can continue to reuse that value so long as the explicit expiration time has not yet passed.

In most cases, a short session can make revocation unnecessary: the user can access the server until the session expires, at which time the server can refuse to issue a new authenticator. Servers that require secure revocation should keep track of the session status on the server (e.g., using a random session key or our personalized scheme with a server database). This session can then be explicitly revoked on the server, without trusting the client.

The scheme does allow simultaneous revocation of all authenticators, which can be accomplished by rotating the server key. This will cause all outstanding cookies to fail to verify. Thus, all users will have to log in again. This might be useful for finding unused accounts.

4.4 Design alternatives

One interesting point of our scheme is that we have included the expiration time t in the cookie value itself.

This is the only way for a server to have access to the expiration date without maintaining state. Explicit inclusion of the expiration date in a non-malleable cookie provides fixed-length sessions without having to trust the client to expire the cookie. It would also have been possible to merely use a session identifier but that would always require server state and might lead to mistakes where expiration was left in the hands of the client.

Many schemes do involve setting a random session identifier for each user. This session identifier is used to access the user's session information, which is stored in a database on the server. While such a scheme allows for a client to make customizations (i.e. it is functionally equivalent to the scheme we have presented), it is potentially subject to guessing attacks on the session identifier space. If an adversary can successfully guess a session identifier, the system is broken (see Section 3.3). Our scheme provides a means for authenticating clients that is resistant to guessing attacks on session identifiers. Furthermore, our scheme provides the option of authenticating clients with $O(1)$ server state, rather than $O(n)$, where n is the number of clients.

Our system can also make it easier to deploy multi-server systems. Using session identifiers requires either synchronized, duplicated data between servers or a single server to coordinate requests, which becomes a potential bottleneck. Our scheme allows any server to authenticate any user with a minimum of information, none of which must be dynamically shared between servers. In addition, the authentication always completes in constant time, rather than in time which increases with the number of users.

5 Security analysis

In this section we present an informal analysis of the security properties of our design. For the purpose of discussion, we will refer to the cookie's two halves: the *plaintext* and the *verifier*. The plaintext is the expiration concatenated with the user string, and the verifier is the HMAC of the plaintext.

We will discuss the security of the scheme once the authenticator (i.e. cookie) is received by the user from the server. We will not discuss mechanisms for completing the initial login.

5.1 Forging authenticators

An adversary does not need to log in if it can create a valid authenticator offline. Often an adversary can create a plausible plaintext string; therefore the security of the authenticator rests on the fact that the verifier cannot be calculated by an adversary without the key. Since we have selected our MAC to be non-malleable, an adversary can not forge a new authenticator.

An attacker may also attempt to extend the capabilities associated with the authenticator. This might include changing the expiration date or some aspect of the data string which would allow unauthorized access to the server. For instance, if the data string includes a username, and the adversary can alter the username, this might allow access another user's account. It is easy enough for the adversary to change the plaintext of the authenticator in the desired manner. However, as we have seen, because HMAC is non-malleable, it is intractable for the adversary to generate a valid ciphertext for an altered plaintext string. Therefore the adversary cannot bring about any change in an authenticator that will be accepted by the server.

5.2 Authenticator hijacking

An interrogative adversary cannot see any messages that pass between the user and the server. Therefore, it cannot hijack another user's authenticator. However, an eavesdropper can see the authenticator as it passes between the user and the server. Such an adversary can easily perform a replay attack. Therefore the system is vulnerable to hijacking by such an adversary. However, the replay attack lasts only as long as the authenticator is valid; that is, between the time the adversary "sniffs" the authenticator and the expiration time. The adversary does not have the ability to create or modify a valid authenticator. Therefore this is an attack of limited usefulness. The lifetime of the authenticator determines how vulnerable the system is; systems which employ a shorter authenticator lifetime will have to reauthenticate more often, but will have tighter bounds on the damage that a successful eavesdropping adversary can accomplish. In addition, the system can protect against an eavesdropping adversary by using SSL to provide confidentiality for the authenticator.

5.3 Other attacks

We mention briefly some attacks on our schemes which do not deal with the authenticator directly. The best known attack against the scheme in Section 4 is a brute force key search.

A server compromise breaks the system: if the adversary obtains the key to the MAC, it can generate valid authenticators for all users. Random keys and key rotation help to prevent the adversary from mounting brute force key attacks (see Lenstra [27] for suggestions on key size).

In addition, key rotation helps protect against volume attacks, whereby an adversary may be able to obtain the key to the hash function because the adversary has obtained a great quantity of data encrypted using it. We note that HMAC-MD5 and HMAC-SHA1 are not believed to be vulnerable to this type of analysis [23]. However, we believe that it is prudent to include key rotation since it does not decrease the security of the scheme, it protects against server compromise, and it has minimal cost to the server.

In addition, the adversary can obtain unauthorized access by guessing the user's password; see Section 3.2 for some guidelines for preventing this.

Our scheme in itself only provides user authentication. For protection against server impersonation or for data integrity, we recommend SSL.

6 Implementation and performance

The client authentication scheme described in Section 4 was implemented in Perl 5.6 using the LWP, HTTP, CGI, FCGI, and Digest modules. We tested the implementation on two dual Pentium III 733 MHz machines each with 256 MB of RAM running the Linux 2.2.18-smp kernel and Apache 1.3.17 with mod_fastcgi 2.2.10. Everything ran on a local disk. A dedicated Gigabit link with a 20 μs round-trip time connected the machines.

6.1 Microbenchmark performance

We ran $1,000$ trials of `crypt()` and HMAC-SHA1. The input to `crypt()` was an 8-byte input and a 2-byte salt. The input to HMAC-SHA1 was a 27-byte input and a 20-byte key. `crypt()` finished on average in 8.08 μsec with 99% of the trials completing in under 10 μsec. HMAC-SHA1 took on average 41.4 μsec with 99% of the trials completing in under 47 μsec. We attribute the variances to context switching.

6.2 End-to-end performance

To measure the end-to-end performance of cookie-based logins, we repeatedly retrieved 400 bytes of data from a Web server that authenticated our client. Both the client and the cookie authentication scheme were implemented in Perl, and the server ran the cookie authentication script with FastCGI. Our end-to-end test consisted of the client presenting a cookie authenticator (as described in Section 4) to the server, which verifies the authenticator by performing HMAC-SHA1 on the expiration date presented by the client. In order to provide a baseline for comparison, we also measured the average performance of plain HTTP, HTTP with Basic Authentication [16], and an always-authenticated FastCGI script for the same page.

For each scheme, we made 5,000 successive requests, with valid authentication information (when needed). Figure 3 presents the average time from the request being sent in our HTTP client until a response was received.

99% of the HTTP trials without authentication were faster than 5.9 ms. Similarly, 99% of HTTP Basic authentication trials were faster than 6.3 ms. 99% of the plain FastCGI trials were faster than 7.7 ms, and 99% of the FastCGI trials with our HMAC-SHA1 scheme took less than 8.8 ms. Figure 3 shows that the cost of HTTP Basic authentication is 0.4 ms per request while the cost of our HMAC-SHA1 scheme is 1.2 ms. We suspect that non-cryptographic factors such as string parsing and file I/O cause the disparity between the microbenchmarks and the end-to-end measurements.

Note that SSL is an order of magnitude slower than the HMAC-SHA1 cookie scheme. A single new SSL connection takes 90 ms [17] on a reasonable machine. SSL client authentication, even with session resumption, cannot run faster than the HMAC-SHA1 cookie scheme because SSL authenticates the entire HTTP stream. Our scheme runs HMAC-SHA1 on fewer than 30 bytes of data per request (a timestamp, personalization data, and a key).

7 Related work

There is an extensive body of work related to authentication in general and Web authentication in particular. We highlight a few relevant examples. For other studies of design principles, see Abadi [1] or Lampson [26].

7.1 General authentication protocols

In the past ten years, several new authentication protocols have been developed, including AuthA [4], EKE [5], provably secure password authenticated key exchange [7], and the Secure Remote Password protocol [44]. Furthermore, groups are simplifying and standardizing password authentication protocols [21]. However, these protocols are not well-suited for the Web because they are designed for session initialization of long-running connections, as opposed to the many short-lived

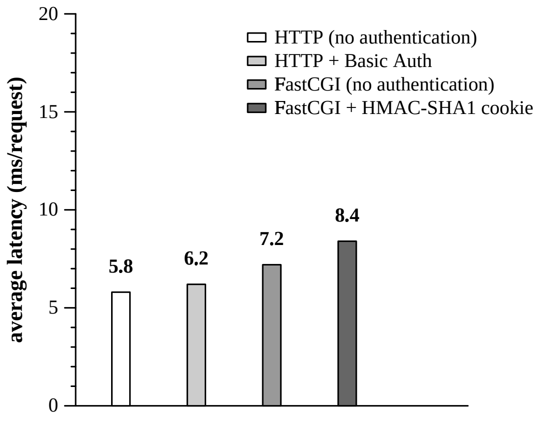

Figure 3: End-to-end performance of average service latency per request. We measure HTTP and FastCGI without authentication to obtain a baseline for comparison. Basic Auth is the cleartext password authentication in HTTP [16].

connections made by Web browsers. Long-running connections can easily afford a protocol involving the exchange of multiple messages, whereas short-lived ones cannot absorb the overhead of several extra round-trips per connection. Additionally, these protocols often require significant computation, making them undesirable for loaded Web servers.

One-time passwords can prevent replay attacks. Lamport's user password authentication scheme defends against an adversary who can eavesdrop on the network and obtain a copies of server state (i.e. the hashed password file) [25]. This scheme is based on a one-way function. Haller later implemented the S/Key one-time password system [19, 20] using techniques from Lamport. De Waleffe and Quisquater extended Lamport's scheme with zero-knowledge techniques to provide more general access control mechanisms [10]. With their one-exchange protocol, a user can authenticate and prove possession of a ticket. This scheme is not appropriate for our model of Web client authentication because it requires the client to perform computation such as modular exponentiation.

Kerberos uses tickets to authenticate users to services [22, 33, 40]. The Kerberos ticket is encrypted with a key known only to the service and the Kerberos infrastructure itself. A temporary session key is protected by encryption. The ticket approach differs greatly from schemes such as ours because tickets are message preserving, meaning that an adversary who compromises a service key can recover the session key. If an adversary compromises the key in our scheme, it can mint and verify tokens, but it cannot recover the contents that were originally authenticated. Authentication and encryption should be separated, but Kerberos does both in one step.

The Amoeba distributed operating system cryptographically authenticated capabilities (or rights) given to a user [42]. One of the proposed schemes authenticated capabilities by XORing them with a secret server key and hashing the result. Client authentication on the Web falls into the same design space. A Web server wishes to send a user a signed capability.

7.2 Web-specific authentication protocols

The HTTP specifications provide two mechanisms for authentication: Basic authentication and Digest authentication [16]. Basic authentication requires the client to send a username and password in the clear as part of the HTTP request. This pair is typically resent preemptively in all HTTP requests for content in subdirectories of the original request. Basic authentication is vulnerable to an eavesdropping adversary. It also does not provide guaranteed expiration (or logout), and repeatedly exposes a user's long-term authenticator. Digest authentication, a newer form of HTTP authentication, is based on the same concept but does not transmit cleartext passwords. In Digest authentication, the client sends a cryptographic hash (usually MD5) of the username, password, a server-provided nonce, the HTTP method, and the URL. The security of this protocol is extensively discussed in RFC 2617 [16]. Digest authentication enjoys very little client support, even though it is supported by the popular Apache Web server.

The main risk of these schemes is that a successful attack reveals the user's password, thus giving the adversary unlimited access. Further, breaks are facilitated by the existence of freely available tools capable of sniffing for authentication exchanges [39].

The Secure Sockets Layer (SSL) protocol is a stronger authentication system provides confidentiality, integrity, and optionally authentication at the transport level. It is standardized as the Transport Layer Security protocol [11]. HTTP runs on top of SSL, which provides all the cryptographic strength. Integration at the server allows the server to retrieve the authentication parameters negotiated by SSL. SSL achieves authentication via public-key cryptography in X.509 certificates [8] and requires a public-key infrastructure (PKI). This requirement is the main difficulty in using SSL for authentication — currently there is no global PKI, nor is there likely to be one anytime soon. Several major certificate authorities exist (e.g., Verisign), but the space is fractured and disjoint. To some degree, users avoid client certificates because certificates are practically incomprehensible to non-technical users. Other arguments suggest that the merits of PKI as the answer to many network security problems have been somewhat exaggerated [13]. Client support for SSL is non-standard and thus can have interoperability problems (e.g., Microsoft Internet Explorer and Netscape Navigator client certificates do not interoperate), and performance concerns. SSL decreases Web server performance and often provides more functionality than most applications need. In an effort to avoid using SSL, Bergadano, Crispo, and Eccettuato use Java applets to secure HTTP transactions [6].

Park and Sandu identify security problems of regular cookies, network threats, end-system threats, and cookie harvesting threats [34]. Samar describes a cookie-based distributed architecture for single-signon [37].

7.3 Schemes in the field

Many ad hoc schemes are used today to perform Web authentication without making use of either SSL or any of the HTTP authentication mechanisms. Instead, schemes often use HTTP state management to store authenticators with the client. This helps sites provide authentication for Web applications while preserving ease-of-use and performance. While many of these schemes are well-designed and do indeed provide appropriately strong authentication for the environment in which they are deployed, just as many schemes have fatal flaws.

Shibboleth, a project of Internet2, is investigating architectures, frameworks, and technologies to support cross-realm authentication and authorization for access to Web pages [38]. The group completed a survey of client authentication on the Web at several universities, most of which use a combination of Kerberos, client certificates, HTTP authentication, and cookies. However, they have not yet presented a complete design.

Open Market has patented a scheme that creates a folded cryptographic hash of a server secret, a session identifier, and other parameters [28]. Yahoo has a cookie authentication scheme that computes MD5 of a server secret, user identifier, timestamp, and other parameters [46]. The ArsDigita Community System (ACS) has a SHA1-based cookie authentication scheme [29]. All these schemes are likely to be secure against interrogative adversaries, but all appear vulnerable to eavesdroppers.

Microsoft Passport offers a managed cookie authentication scheme [35]. Microsoft mints a cookie authenticator after a user logs in. Vendors participating in the passport service can verify the authenticator to determine authenticity and authorization. The details of the authentication scheme have not been published, but the white paper indicates that Microsoft shares a unique symmetric key with each vendor. These keys can both mint and verify authenticators.

8 Conclusion

To provide designers and implementers with a clear framework, we have given a description of the limitations, requirements, and security models specific to Web client authentication. We presented a set of hints on how to design a secure client authentication scheme, based on experience gained from our informal survey of commercial schemes. The survey showed that many sites are not secure against the interrogative adversary. We proposed an authentication scheme secure against the interrogative adversary.

Web sites have such a large range of requirements that no one authentication scheme can meet them all. Currently SSL remains too costly and client authentication infrastructures remain hardly deployed. This partially explains why so many home-brew schemes exist. The Web community ought to recommend a secure standard or secure practices if there is any hope to eliminate the proliferation of insecure home-brew authentication schemes. We hope that this paper will help schemes in resisting common attacks.

For more information and our source code, see the appendix or visit our Web site at `http://cookies.lcs.mit.edu/`.

9 Acknowledgments

We thank David Andersen, Ian Anderson, Jeffrey W. Baker, Richard Barbalace, Andrew M. Boardman, Benjie Chen, David Dittrich, Paul Hill, Frans Kaashoek, David Mazières, Robert T. Morris, Steve Morris, Joon Park, Matt Power, Ron Rivest, Jerry Saltzer, Richard Smith, Win Treese, the anonymous reviewers, and the members of the PDOS group at MIT. We also thank the companies who talked with us about the security of their Web sites: `FatBrain.com`, `WSJ.com`, and `yahoo.com`. The students of the MIT Applied Security Reading Group (`http://pdos.lcs.mit.edu/asrg/`) deserve credit for the genesis of this project. Finally, we thank Duncan Hines for manufacturing the materials necessary to sustain our efforts.

References

[1] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. Technical Report 125, DEC Systems Research Center, June 1994.

[2] Allaire Corporation. Personal Communication, January 2001.

[3] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *Proceedings of Advances in Cryptology—CRYPTO 98*, volume 1462 of *Lecture Notes in Computer Science*, pages 26–45, Santa Barbara, CA, 1998. Springer-Verlag.

[4] Mihir Bellare and Phillip Rogaway. The AuthA protocol for password-based authenticated key exchange. Technical report, IEEE P1363, March 2000. `http://grouper.ieee.org/groups/1363/StudyGroup/Passwd.html#autha`.

[5] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 72–84, Oakland, CA, May 1992.

[6] F. Bergadano, B. Crispo, and M. Eccettuato. Secure WWW transactions using standard HTTP and Java applets. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 109–119, Boston, MA, September 1998.

[7] Victor Boyko, Philip MacKenzie, and Sarvar Patel. Provably secure password authenticated key exchange using Diffie-Hellman. In B. Preneel, editor, *Proceedings of Advances in Cryptology—EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, Bruges, Belgium, May 2000. Springer-Verlag.

[8] CCITT. Recommendation X.509: The directory authentication framework, 1998.

[9] CERT. Malicious HTML tags embedded in client Web requests. CA-2000-02, February 2000. `http://www.cert.org/advisories/CA-2000-02.html`.

[10] Dominique de Waleffe and Jean-Jaques Quisquater. Better login protocols for computer networks. In B. Preneel, R. Govaerts, and J. Vandewalle, editors, *Proceedings of Computer Security and Industrial Cryptography*, volume 741 of *Lecture Notes in Computer Science*, pages 50–70. Springer-Verlag, 1993.

[11] Tim Dierks and Christopher Allen. The TLS protocol version 1.0. RFC 2246, Network Working Group, January 1999.

[12] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. In *Proceedings of the* 23*rd ACM Symposium on Theory of Computing*, pages 542–552, New Orleans, LA, 1991.

[13] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[14] Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, Network Working Group, June 1999.

[15] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[16] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, Network Working Group, June 1999.

[17] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, CA, October 2000.

[18] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[19] Neil Haller. The S/KEY one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, pages 151–157, San Diego, CA, February 1994.

[20] Neil Haller. The S/KEY one-time password system. RFC 1760, Network Working Group, February 1995.

[21] IEEE P1363a: Standard specifications for public key cryptography: Additional techniques. `http://www.manta.ieee.org/groups/1363/P1363a`.

[22] John T. Kohl. The use of encryption in Kerberos for network authentication. In G. Brassard, editor, *Proceedings of Advances in Cryptology—CRYPTO 89*, volume 435 of *Lecture Notes in Computer Science*, pages 35–43. Springer-Verlag, 1990.

[23] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Network Working Group, February 1997.

[24] David Kristol and Lou Montulli. HTTP State Management Mechanism. RFC 2965, Network Working Group, October 2000.

[25] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–771, November 1981.

[26] Butler Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 33–48, Bretton Woods, NH, 1983.

[27] Arjen Lenstra and Eric Verheul. Selecting cryptographic key sizes. `http://www.cryptosavvy.com/cryptosizes.pdf`, November 1999.

[28] Thomas Levergood, Lawrence Stewart, Stephen Morris, Andrew Payne, and Winfield Treese. Internet server access control and monitoring systems. U.S. patent #5,708,780, Open Market, January 1998.

[29] Richard Li and Archit Shah. ArsDigita Community System (ACS) security design. `http://developer.arsdigita.com/doc/security-design.html`.

[30] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 1997.

[31] Keith Moore and Ned Freed. Use of HTTP State Management. RFC 2964, Network Working Group, October 2000.

[32] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):584–597, November 1979.

[33] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[34] Joon S. Park and Ravi Sandhu. Secure cookies on the Web. *IEEE Internet Computing*, 4(4):36–44, July/August 2000.

[35] Microsoft passport. `http://www.passport.com/`.

[36] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[37] Vipin Samar. Single sign-on using cookies for Web applications. In *Proceedings of the 8th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 158–163, Palo Alto, CA, 1999.

[38] The Shibboleth Project. `http://middleware.internet2.edu/shibboleth/`.

[39] Dug Song. *dsniff*. `http://www.monkey.org/~dugsong/dsniff/`.

[40] Jennifer Steiner, Clifford Neuman, and Jeffrey Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*, pages 191–202, Dallas, TX, February 1988.

[41] Paul Syverson, Stuart Stubblebine, and David Goldschlag. Unlinkable serial transactions. In R. Hirschfeld, editor, *Proceedings of Financial Cryptography*, volume 1318 of *Lecture Notes in Computer Science*, Anguilla, BWI, 1997. Springer-Verlag.

[42] Andrew Tanenbaum, Sape Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing*, pages 558–563, Cambridge, MA, 1986.

[43] David Wagner and Ian Goldberg. Proofs of security for the Unix password hashing algorithm. In T. Okamoto, editor, *Proceedings of Advances in Cryptology—ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, Kyoto, Japan, December 2000. Springer-Verlag.

[44] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[45] Web and mobile code security. `http://www.securityfocus.com/forums/www-mobile-code/`.

[46] Yahoo, Inc. Personal Communication, November 2000.

A Search engine queries

Cookie files are occasionally published on the Internet and are indexed by some search engines. Variants of these queries have at times worked on `Google.com`, `Yahoo.com`, `NorthernLight.com`, and `AltaVista.com`.

After we reported these queries, Google immediately removed all files called "`cookies.txt`" or "`COOKIES.TXT`" from their indexing and later their cache. This removes the indexing for most cookie files. Some cookie files still exist under non-standard file names, apparently because of corrupted text files (e.g., resumes that include a person's cookie file at the end). The other search engines gave no definitive responses. For historical purposes, here are the queries that used to produce many cookie files.

- avenuea.com FALSE FALSE
- CERT7.DB
- text:CERT7.DB

The idea of the search queries is to locate cookie files based on information inside the cookie. For instance, `avenuea.com` is found in most cookie files because of online advertising and user tracking. CERT7.DB often appears near files called `cookies.txt`. Censoring `cookies.txt` will not prevent someone from searching for CERT7.DB then indirectly finding a cookie file.

B ichat Rooms v3.0

ichat Rooms v3.0 (`www.ichat.com`) is a Web-based commercial chat server system. A user logs in with a username/password and receives a cookie, which is composed of the username and password XORed with a universal constant value. Subsequent requests are authenticated with this cookie.

This system is vulnerable to an attack by an eavesdropping adversary. The adversary can replay the cookie since it never expires. Knowledge of the constant string (easily obtained by an interrogative adversary) allows an adversary to reconstruct the user's password as well. Figure 4 contains a sufficient number of sample cookies to recover the constant key.

We did not analyze newer versions of ichat Rooms. We only know that the current scheme is different.

C New England Bride (nebride.com)

`nebride.com` is the Web site for the New England Bride magazine. A user logs in with a username/password and receives an ID cookie, which is simply a username. This cookie authenticates subsequent requests.

An interrogative adversary can achieve a selective forgery by guessing the username of an existing account on the server and use this for full access to the account. Furthermore, since the password is displayed in plaintext on the server's information page, the adversary can retrieve the victim's password, possibly compromising accounts of the same user on other systems.

We notified New England bride, but we are not sure if they understand the problem.

D Fatbrain.com

A user logs in with their email address and password, then receives a URL-based authenticator from the server. The URL-based authenticator is generated from a global, monotonically increasing sequence number. The URL-based authenticator is used to authenticate the remainder of the user's session.

This attack can proceed more quickly if the victim has logged in recently, or the time at which the user last logged in can be estimated. In that case, an adversary can begin searching backwards in the sequence space for the proper authenticator. Using this authenticator, an adversary can change the victim's email address, receive email informing them of the victim's password, and thereby gain full control — including purchasing ability — over the victim's account.

The `Fatbrain.com` pages use an easily predictable sequence number as an authenticator. This permits an interrogative adversary to guess the authentication token.

The engineers at Fatbrain responded swiftly by changing the sequence number to a random number. This defeats the attack described above.

E ihateshopping.net

A user logs in with a username/password and receives an ephemeral ID cookie, which is simply a serially-assigned integer unique to the user. This cookie is used to authenticate the user's requests for the remainder of

username	fubob
password	aaaaaaaa
cookie	ichat_cookie=0F160A0E1656233E21254D080B0B0D0058080B5359
username	fubbb (note the 1 letter change)
password	aaaaaaaa
cookie	ichat_cookie=0F160A031656233E21254D080B0B0D0058080B5359

Figure 4: Sample ichat authenticators.

username	fubob
cookie	ID=fubob

Figure 5: Sample nebride.com authenticator

that session. An interrogative can access arbitrary accounts because of the densely-populated namespace of ID cookies.

The ihateshopping.net pages provide a simple numeric authenticator, the namespace of which can be trivially stepped through to reveal the account details of all accounts on the system. The adversary would not require their own account or any other sort of prior access.

We notified the site and received a quick response. We do not know how the current authentication scheme differs because the Web site appears to no longer exist.

F SprintPCS.com

A user logs in with their phone number and account password (PIN), then receives a cookie from the server which appears to be some form of the standard Unix crypt() function applied to the submitted data. The cookie provides administrative access to one's account, and this and subsequent account-related transactions are protected with SSL. The cookie's "SSL Only" flag, however, is not set, and the cookie domain is sprintpcs.com.

An eavesdropping adversary can obtain a user's authentication cookie if the user logs in to the secure section of sprintpcs.com and later revisits a non-SSL section of the site. Since the cookie is used for authentication only within SSL-protected sections of the sprintpcs.com site, setting the "SSL Only" flag would provide increased protection for the authenticator.

While SprintPCS appears to spend many CPU cycles on SSL, the SSL is of little benefit since the secret leaks in plaintext.

We notified SprintPCS through their Web interface.

G Snowball.com Web sites

A user registers with a username/password and receives an authentication cookie. The site uses this cookie to authenticate the user for future logins. The same scheme is used on chickclick.com, highschoolalumni.com, and ign.com.

Simply by changing the username and user ID (UID) in the Beacon cookie to the that of the victim, an adversary can login as the victim. First, an adversary logs in to obtain a legitimate username/UID pair included in the Beacon cookie. Given such a cookie, the adversary changes the username and UID to that of the victim. Username and UID information are readily available from the Web site itself. In the case of highschoolalumni.com, knowing a user's high school enables an adversary to discover the username and UID of a user.

This is an example of a non-cryptographic authentication scheme. We suppose that the site wanted to reduce database lookups for each request.

We notified snowball.com and eventually received a response. We have not inspected the site recently.

H Yahoo.com

Yahoo provided us with a description of their client authentication scheme.

Architecture Overview

Yahoo provides a large number of services with essentially a single sign-on. In the most basic case, each user has a username, a unique user ID (that may change over https://fatbrain.com/...?t=0&p1=fubob@mit.edu&p2=540555758

Figure 6: Sample `Fatbrain.com` authenticator

RM%5FON=Y&CN1=X&R115=Y

Figure 7: In the `SprintPCS.com` cookie, X and Y represent 13-character `crypt()`-like outputs. R115 is a function of the user's password. CN1 is likely a function of the user's phone number.

time), and a password. For more secure services (such as Yahoo! Wallet), an additional secure key (a longer password) is required as well. Authentication and personal information is encoded into a small set of cookies.

The overall architecture consists of:

- a central user database, which maps a user's ID to a long list of properties,
- some login machines, such as `login.yahoo.com`,
- all the service machines, like `my.yahoo.com` or `calendar.yahoo.com`.

HTTP and HTTP over SSL are used to communicate between clients and Yahoo.

Authentication Scheme

Yahoo's authentication scheme uses cookies to store user and authentication information on the user's computer. Users are initially authenticated via password which can be performed either over regular HTTP or HTTP layered on SSL. This allows users to protect their password from passive adversaries if they so choose. All logins must be processed initially via the login machines.

Upon successful login, users are given 2 authentication cookies at Yahoo: Y and T. Y is persistent, but T is normally lost when the browser exits. The persistence of the Y cookie can be configured by the user to one of a fixed set of expiration times ranging from 15 minutes to one day.

The Y cookie alone is accepted for some low-security things, like displaying the user's My Yahoo page. For more "personal" info, like Calendar or Mail, an up-to-date T cookie is also needed. The Y cookie encodes the user's userid, some basic demographic info, and the user's internal unique ID. For instance, a test user ID "tlbtlbtlb2" gets the following in Figure 9.

The 'l' part is a transposed version of the ID. The 'n' part is that login's unique ID. It's compared to the version stored in the central database when querying to retrieve any information.

Other parameters encode language/country preferences, gender, year of birth, and regional information.

These allow certain simple content customizations to happen quickly — for example, gender is for targeting ads, year of birth helps filter out adult from search results to kids, and zipcode can be used for weather and yellow pages search. Also, encoding this directly helps reduce load on the database machines since service machines do not need to query for this common information specifically.

The T cookie is basically a timestamp and a symmetric digital signature. A sample T cookie is in Figure 10.

The 'z' is for backwards compatibility. The 'a' field contains flags that specify the expiration time and some flags related to protecting privacy of minors. The important fields are 'd' and 'sk', which are a timestamp and a signature on the login, unique ID, and timestamp. (The 'd' field may also include some other things for backwards compatibility.) The signature does require having a Yahoo-wide shared secret that every machines knows. This signature is calculated as an MD5 hash over the data with the shared key. This scheme also relies upon a loosely synchronized clock, much like Kerberos does.

For maximum security parts of the system, like Yahoo! Wallet, yet another cookie, called the S cookie, is used. The S cookie is actually very different in structure as the generation and management of the contents is different. This cookie is limited to machines in the `.secure.yahoo.com` domain, and only in SSL mode. It is roughly like the T cookie in structure. This cookie is issued when you enter your Yahoo! security key, which is only entered in SSL mode, and which has some password quality rules enforced. The secure S cookies, and the vendor-specific cookies that we generate

Beacon=`hsareg.uid.username.hsa0.976659917`

Figure 8: A sample `Snowball.com` cookie

Y=v=1&n=5qhie84hrpd9n&l=jb1jb1jb1s/o&p=m252rq8401b304&r=3k&lg=us&intl=us&np=1

Figure 9: Yahoo Y cookie

T=z=eFGF6AeLbF6Acbnur1trzJ7&a=gEE&sk=DAAkjV.h3/r2GU&d=YQFnRUUBenoBZUZHRjZBZ1dB

Figure 10: Yahoo T cookie

from them, are tied to a particular IP address. Although most HTTP proxies make the IP address of users jump around, Yahoo reports that this appears to be quite rare in the HTTPS case, so the IP address check doesn't seem to hurt anyone.

By limiting this cookie to the `.secure.yahoo.com` domain, Yahoo ensures that only a carefully administered set of machines can access this cookie. These systems present very simple user interfaces (just the password entry pages) to minimize the chance of allowing some Javascript cookie stealing hack.

Some services hosted by Yahoo, like Stores and Paydirect, require access to secure data but don't run under `.secure.yahoo.com`. Such requests are authenticated via a separate system. Initial requests requiring authentication are redirected to a `.secure.yahoo.com` machine which requests the secure password. Upon valid authentication here, the users are redirected back to the hosted service with an authentication token on the end of the URL. The original machine then sets a cookie similar to the S cookie that is visible only to it and authenticated using a secret key that is private to the original machine. When that cookie expires (according to the included timestamp), the service will have to re-send the client to the `.secure.yahoo.com` machines for reauthentication. The URL authenticator is service-specific and is also checked for replays. This ensures that a sniffed authenticator is not useful to any adversary.

Yahoo validates the integrity of all data sent in the T (and presumably S) cookies, except for the demographics information used for ad targeting.

Invalidation and revocation are handled by changing the unique ID in the user database (while preserving the username and other associated information). By ensuring that the unique ID is changed whenever the password is changed, Yahoo ensures that any outstanding cookies are immediately invalidated.

Discussion

The HTTP scheme appears secure against the interrogative adversary, and the HTTP over SSL scheme against the active adversary. However, the scheme may be vulnerable to cryptanalytic splicing attacks because it does not use a strong keyed MAC, but instead uses straight MD5 with a fixed shared key. This construction does not offer the same amount of confidence as HMAC-MD5 might provide. Fortunately, unlike `crypt()`, MD5 considers the entire input, making an attack like the one executed against the Wall Street Journal much less likely.

The use of a single shared key is also a risk because compromise of this shared key could compromise the same portion of Yahoo's system.

Remedy

There is no known attack against this scheme as of yet, except that the use of the hash as a MAC is not known to resist splicing attacks.

I WSJ.com

The fastlogin cookie is an authenticator issued to a user after typing in a username/password on `WSJ.com`. The algorithm was determined to be:

fastlogin =
username + crypt (username + rotating server secret)

where + denotes concatenation without delimiters.

This scheme is weaker than schemes such as HTTP authentication which send cleartext passwords over the network. An interrogative adversary who discovers the algorithm can forge a cookie authenticator for any user. This results in a total break.

Discussion

The vulnerability on the WSJ.com site gives an adversary access to any user's subscription and personal information. In addition, an adversary can purchase items at the WSJ and affiliated sites (such as archived articles) under any user's credit card. Furthermore, an adversary can view optional information set by the user. For instance, many users keep a list of their stock portfolio on the WSJ.com site.

An adversary needs only one piece of information to gain access: the username of the victim. In this document we explain nine security holes on WSJ.com.

1. With knowledge of a username, an adversary can log in as that user.

   The WSJ.com site requires a paid account to read articles, purchase archived articles, etc. A user can log into a personalized WSJ.com site if the user's Web browser has a valid "fastlogin" cookie. Because of several mistakes in the use of cryptography, we were able to write a program that, given a username, creates a working fastlogin cookie. This program is attached at the bottom of this document.

   Figure 1 shows an example of a fastlogin cookie.

   The last field represents the username, bitdiddle, prepended to the output of the Unix crypt() function. The input to the crypt() function is the username prepended to the string "March20".

   Likely WSJ.com expected the fastlogin cookie to act like a one-way hash or something that depends on secret information. However, the fastlogin cookie is a deterministic value which can always be computed from just the first 8 characters of the username.

2. If an adversary is not aware of vulnerability # 1, the adversary can still access most accounts.

   Before discovering vulnerability #1, we considered a less serious attack that allowed access to most WSJ.com accounts. Any curious person with two accounts having similar usernames could notice this vulnerability. However, vulnerability #1 contains all of the implications of vulnerability #2 and more.

   The crypt() function only pays attention to the first 8 characters of its input. The implication is that for all usernames that match in the first eight characters, the fastlogin cookie is the same. Everything after the 8th character of a username is ignored.

   This attack works against all accounts that have 8 to 14 character usernames. Only users with 5, 6, or 7 character usernames are safe from this simple attack. However, all users are still susceptible to vulnerability #1.

   In other words, if a victim has the username "bitdiddle", then an adversary can register for another account with the victim's username as the prefix (e.g., "bitdiddler"). This results in the same crypt() output:

username	Crypt() Output	Fastlogin Cookie
bitdiddle	MaRdw2J1h6Lfc	bitdiddleMaRdw2J1h6Lfc
bitdiddler	MaRdw2J1h6Lfc	bitdiddlerMaRdw2J1h6Lfc

   If an adversary wishes to take control of the account of another user, the adversary must only know the login name of the victim. The adversary then creates a new user on WSJ.com such that the new username starts with the victim's username (e.g., "bitdiddler"). The adversary edits the cookie to remove the "r" in "bitdiddler". Next, the adversary starts a Web browser (e.g., Netscape) and goes to www.wsj.com. The site allows the adversary to log in as the victim. Now the adversary has access to the victim's credit card and can view personal information such as home phone numbers and addresses. In addition, the adversary can change the victim's password, although this is not necessary. The only condition is that the victim's username must be at least 8 characters for the incorrect use of crypt() to appear. Again, this vulnerability is eclipsed by vulnerability #1 mentioned earlier.

   The implication scenarios are the same as vulnerability #1.

3. The salt is constant.

   WSJ.com uses the same salt, "Ma", for every user. The salt is supposed to help randomize the output of crypt(). When WSJ.com issues fastlogin cookies, it ought to set a random 2-character salt rather than simply "Ma" for every user.

   Because the fastlogin cookie does not take passwords into account, this does not appear to further weaken the scheme. But if WSJ.com were to hash passwords, the constant salt would make dictionary attacks easier to mount against stolen cookies.

4. The secret padding is partially revealed by the salt.

   The salt is not intended to be a private value; it is by definition a public value that is sent along with the ciphertext. Using the same value for a secret string (the padding) and a public string (the salt) is dangerous, because it inherently compromises the secret string. The salt "Ma" consists of the first two

domain	Javascript?	Path	SSL?	Expiration	Variable name	Value
.wsj.com	FALSE	/cgi	FALSE	941452067	fastlogin	bitdiddleMaRdw2J1h6Lfc

Figure 11: A sample cookie authenticator from `WSJ.com`.

characters of the secret padding string "March20" which is used to create fastlogin cookies.

The implication is that the salt partially gives away the secret used to create fastlogin cookies.

We later discovered that the secret string was intended to be a rotating key. However, the rotation was not implemented. The rollout day, "March20", remained the secret key until recently.

5. Lack of secret information and ignored input

   The authentication scheme relies on the secrecy of a 7-character string stored on the `WSJ.com` Web server. We were able to extract this 7-character string.

   We first ran offline a brute force guessing program to determine 3 of the seven characters. This took about an hour on ten 733MHz machines. Then we improved the program by letting it interactively query the `WSJ.com` Web server (about 128×8 times) to determine the remaining 4 characters. Excluding the programming time, this second test took less than 18 minutes of computation (1 second per query because we did not want to flood the server with requests).

   Second, the crypt() function ignores all input after the 8th character. `WSJ.com` ought to use a transform that uses all of its input. For instance, SHA-1 takes an arbitrarily long string and produces a 20-byte output.

   Because of these two flaws, we were able to quickly recover this semi-secret, "March20". This is used as a secret to create fastlogin cookies. The implication is that cryptanalysis is straightforward and fast.

6. Lack of revocation

   Even if a user changes his/her password, the fastlogin cookie remains the same. This prevents the `WSJ.com` site from revoking a compromised account.

   The implication is that stolen cookies are not revocable. Even if a victim changes his or her password, the adversary can reuse an old stolen fastlogin cookie.

7. The fastlogin cookie lasts forever

   There is no cryptographically strong lifetime in the fastlogin cookie. Although `WSJ.com` sets the cookie to expire 11 hours later, a savvy user can modify the cookies file to delay the expiration time indefinitely. Our cookies still worked after 5 days. One way to fix this is to include a timestamp in the message authentication code to enforce a cookie lifetime.

8. Patterns in encrypted text

   `WSJ.com` sometimes uses a second cookie called WSJIE_LOGIN. This appears to be some function of the username and password. However, the encryption scheme in the WSJIE_LOGIN cookie exhibits too many patterns to be secure. We identify ciphertext patterns below. A real encryption or hash algorithm would result in random-looking output. WSJIE_LOGIN is far from random. For example, the pairs below are correlated to the alphabet listed backwards. For instance, 'a' encrypts to 'v', 'b' encrypts to 'u', 'c' encrypts to 't', etc. However, some encryptions appear to vary by username.

Password	Encrypted Password
a	v KfAnAfOi
b	u KfAnAfOi
c	t KfAnAfOi
...	...

   [Spaces added for clarity.]

   The implication is that an adversary may be able to retrieve a password from this cookie.

9. `WSJ.com` allows invalid accounts

   One can use non-existent usernames to log in to read content on `WSJ.com`. While this does not affect any particular user, it allows adversaries to read the WSJ for free. For instance, the cookie in Figure 9 works fine even though it has fewer than 5 characters.

Sample authenticator

Cookie: fastlogin=bitdiddleMaRdw2J1h6Lfc

The program in Figure 13 will generate a cookie authenticator given a username.

```
.wsj.com   TRUE / FALSE  1314159265    fastlogin    abcdMaTFoOb31s/Gg
```

Figure 12: A cookie for a non-existent user where username = “abcd” and crypt() = “MaTFoOb31s/Gg”.

```
#!/usr/bin/perl
$salt = "Ma";
$pad = "March20";

print STDERR "Enter username: ";
$username = <STDIN>;
chop $username;

$in = $username . $pad;
while (length ($in) > 8) { chop $in; }

$fasterlogin = crypt ($in, $salt);

print STDERR "Place the following in your .netscape/cookies file.\n\n";
print ".wsj.com TRUE    /       FALSE   1314159265      fastlogin       $username$fasterlogin\n";
```

Figure 13: A program to create `WSJ.com` cookie authenticators.

We used dynamic programming in an adaptive chosen message attack to recover the rotating secret server key, “March20”, in Figure 14. The program runs in $128 * 8$ queries rather than the intended 128^8 (1,024 vs. 72,057,594,037,927,936). Assuming each query takes 1 second, this program finishes in 17 minutes instead of the intended $2 * 10^9$ years. The rotating secret was supposed to be the current date, but the secret got stuck on the rollout date, March 20.

Remedy

We met with Dow Jones, the parent company of the Wall Street Journal, shortly after discovering the vulnerability. The cookie authentication scheme was immediately changed. However, we have not investigated the new scheme. The people at Dow Jones were extremely responsive and helpful.

```
#!/usr/bin/perl
use LWP::UserAgent;
use HTTP::Cookies;

$mysalt = "Ma";  # The well-known 2-char salt
$url    = "http://interactive.wsj.com/pages/money.htm";  # URL returning 200 only if cookie ok
$cookiefile = "/tmp/.netscape/cookies.txt";
$ua = new LWP::UserAgent;
$ua->agent("Cookie-Eaters/1.0");
$request = new HTTP::Request ('GET', $url);
$cookie = HTTP::Cookies::Netscape->new (
            File     => $cookiefile,
            AutoSave => 0, );
$username = "bitdiddl";  # Start with 7-character username to find the left-most padding char
$pad = "";   # What we know about the padding appended to the input of crypt.
$iteration = 1;  # Try every character for the current pad character
for ($count = 0; $count <= 127; $count++) {
    $guess = sprintf("%c", $count);
    $out = crypt ($username . $pad . $guess, $mysalt);
    $cookie->set_cookie(1, "fastlogin" => "$username$out", "/", ".wsj.com");
    $cookie->add_cookie_header($request);
    $ua->cookie_jar( $cookie );
    $response = $ua->simple_request( $request );
    if ($response->is_success) {
        $pad = $pad . $guess;
    }
    if ($iteration == 8) {
        print "Exhausted.  Pad is so far: $pad\n";  exit;
    }
    $iteration++;  sleep 1;
}
```

Figure 14: An adaptive chosen message attack to quickly recover the `WSJ.com` server secret.