Web Security
Secure Socket Layer (SSL)

Web Security

- authentication: basic, digest
- often supplemented by *cookies*
- access control via network addresses
- multi-layered:
  - SHTTP (secure HTTP) = just for HTTP (shttp://)
    CommerceNet, Mosaic
  - SSL ($\rightarrow$ TLS) = generic for TCP (https://)
    implementation: SSLeay
  - IP security: host-to-host

Web vulnerabilities

`http://www.w3.org/Security/Faq/`

Risks:

1. revealing private information on server
2. intercept of client information (credit card records)
3. information about host ➠ break in
4. execute programs, denial of service
5. server log privacy

Web vulnerabilities: information leakage

- Altavista search for `etc/passwd`
- directory listings
- `chroot`
- soft links
- file ownership: local protection $\leftrightarrow$ web access

Web vulnerabilities: cgi-bin

cgi-bin, server-side includes (= macros within HTML)

- server must run at root (port 80!), but executes as “nobody”, “www”, . . .
- cgi-bin: random arguments
- use perl “taint” mode: can’t use variables from environment, standard input, command line for eval(), system(), exec() or piped open()

HTTP access control - basic

- client doesn’t know which method
- client attempts access (GET, PUT, . . .) normally
- server returns

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="WallyWorld"
```

- realm: protection space
- client tries again with

```
Authorization: Basic base64(user:password)
```

- passwords in the clear ➠ not secure
- repeat cycle on each access

HTTP access control - digest

RFC 2069
First attempt for `http://www.nowhere.org/dir/index.html`:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="testrealm@host.com",
                  domain="/dir, /foo",
                  nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                  opaque="5ccc069c403ebaf9f0171e9517f40e41",
                  algorithm=MD5
```

Browser prompts for username (Mufasa) and password (CircleOfLife), retries:

```
Authorization: Digest username="Mufasa",
               realm="testrealm@host.com",
               nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
               uri="/dir/index.html",
               response="e966c932a9242554e42c8ee200cec7f6",
               opaque="5ccc069c403ebaf9f0171e9517f40e41",
               digest="5ccd067f313ebaf9f0171e9517f40e41"
```

HTTP access control - digest

`WWW-Authenticate` parameters:

realm: displayed to user

domain: URIs, remembered by client

nonce: opaque to client (hex, base64, . . .); new for each 401 response
e.g., H(client-IP : time-stamp : server-secret)
can be calculated by server without keeping state

opaque: returned unchanged by client

algorithm: digest, checksum ➠ MD5

HTTP access control - digest

`Authorization` response:

- same nonce, opaque data
- KD(secret, data) = H(secret | : | data)
- A1 = user:realm:password
- A2 = method:uri

  response: H(H(A1) : nonce : H(A2))

  digest: H(H(A1) : nonce : method : data : info : H(body))
  where info = H(uri : type : length : coding : modified : expires)

- request digest useful for POST and PUT
- server only needs H(A1) [protect!], not password
- steal H(A1) ➠ only for realm

HTTP access control - digest

returned with successful request:

- `AuthenticationInfo: nextnonce=...; digest=...`
- avoids 401 failure next time
- also: digest of HTTP body
- subject to man-in-the-middle attack by proxy
- hash is sufficient to gain access (but only one)
- want unique realms
- client can’t authenticate server

Web Server Access Configuration

`http://hoohoo.ncsa.uiuc.edu/docs/tutorials/user.html`

For NCSA `httpd`, `Apache` ➠ `.htaccess` per directory or global:

```
AuthType Basic
AuthUserFile /etc/passwd
AuthName "Private information"

<Limit GET>
order deny,allow
require user hgs
deny from all
allow from .ncsa.uiuc.edu
</Limit>
```

Can reuse /etc/passwd – bad idea (why?)

Web server configuration

Global configuration file `access.conf`:

```
<Directory /full/path/to/protected/directory>
   AuthName           name.of.your.server
   AuthType           Basic
   AuthUserFile       /usr/local/etc/httpd/conf/passwd
   <Limit GET POST>
     require user foo
   </Limit>
</Directory>
```

Web server access configuration

Address-based restrictions:

```
<Limit GET POST PUT>
order deny,allow
deny from all
allow from .cs.columbia.edu
</Limit>
```

is different from

```
<Limit GET POST PUT>
order allow,deny
deny from all
allow from .cs.columbia.edu
</Limit>
```

➠ nobody can use it!

SSL Overview

here: SSL 3.0 ≈ TLS (RFC 2246)

- *secure channel*
- any TCP-based protocol: HTTP (https://, port 443), NNTP, telnet, telephony signaling, . . . ➠ secure byte stream
- optional (but common) public key server authentication
- optional client authentication
- hash: combined MD5 and SHA
- encryption optional (session key), but default: DES, RC2, RC4
- now: TLS (IETF WG)

SSL Cipher Suites

- Diffie-Hellman key exchange
- RSA (see “One-Way Public Key Based Authentication”, 9.3.3)
- Fortezza
- RC2, RC4, 3DES, DES40

SSL Basics

Layered protocol:

1. fragment data into blocks $\leq 2^{14}$ bytes
2. compress data
3. apply message authentication code (MAC) = $H(m|s)$ for message m and secret s
4. encrypt with client (cw) or server (sw) write key
5. transmit over TCP

stateful ➠ handshake to set up keys, algorithms

SSL Messages

Alert	security breach or failure
ApplicationData	actual information
Certificate	sender’s public key
CertificateRequest	client, please send certificate
CertificateVerify	know private key
ChangeCipherSpec	use agreed-upon security service
ClientHello	want, can do
ClientKeyExchange	client’s keys
Finished	negotiations finished
HelloRequest	client, please start negotiation
ServerHello	server capabilities
ServerHelloDone	server done
SererKeyExchange	server’s keys

SSL Data Structures

```
enum {
  change_cipher_spec(20), alert(21), handshake(22),
  application_data(23), (255)
} ContentType;

struct {
  ContentType type;
  ProtocolVersion version;
  uint16 length;
  opaque fragment[SSLPlaintext.length];
} SSLPlaintext;

struct {
  ContentType type;
  ProtocolVersion version;
  uint16 length;
  opaque fragment[SSLCompressed.length];
} SSLCompressed;

block-ciphered struct {
  opaque content[SSLCompressed.length];
```

```
  opaque MAC[CipherSpec.hash_size];
  uint8 padding[GenericBlockCipher.padding_length];
  uint8 padding_length;
} GenericBlockCipher;

digitally-signed struct {
  select(SignatureAlgorithm) {
    case anonymous: struct { };
    case rsa:
      opaque md5_hash[16];
      opaque sha_hash[20];
    case dsa:
      opaque sha_hash[20];
  };
} Signature;
```

SSL Handshake

* = optional
plaintext up to `Finished`!

client		server
	⟵	`HelloRequest*`
`ClientHello`	⟶	
	⟵	`ServerHello`
		`Certificate`
		`ServerKeyExchange*`
		`CertificateRequest*`
	⟶	`ServerHelloDone`
`Certificate*`	⟶	
`ClientKeyExchange`		
`CertificateVerify`		
$[\texttt{Finished}]_{cw}$	⟶	
	⟵	$[\texttt{Finished}]_{sw}$
$[\texttt{ApplicationData}]_{cw}$	⟷	$[\texttt{ApplicationData}]_{sw}$

SSL Handshake

```
enum {
  hello_request(0), client_hello(1), server_hello(2),
  certificate(11), server_key_exchange (12),
  certificate_request(13), server_hello_done(14),
  certificate_verify(15), client_key_exchange(16),
  finished(20), (255)
} HandshakeType;

struct {
  HandshakeType msg_type;    /* handshake type */
  uint24 length;             /* bytes in message */
  select (HandshakeType) {
    case hello_request: HelloRequest;
    case client_hello: ClientHello;
    case server_hello: ServerHello;
    case certificate: Certificate;
    case server_key_exchange: ServerKeyExchange;
    case certificate_request: CertificateRequest;
    case server_hello_done: ServerHelloDone;
    case certificate_verify: CertificateVerify;
    case client_key_exchange: ClientKeyExchange;
    case finished: Finished;
  } body;
} Handshake;
```

Client Hello

$C \rightarrow S$: establish security enhancement capabilities

- random challenge, algorithms supported
- server chooses encryption, compression algorithms

```
struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
struct {
  ProtocolVersion client_version;
  Random random;
  opaque SessionID<0..32> session_id;
  CipherSuite cipher_suites<2..2^16-1>;
  CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

Server Hello

$S \rightarrow C$:

- acknowledges algorithms
- establishes random connection identifier

```
struct {
  ProtocolVersion server_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
} ServerHello;
```

Server Certificate

$S \rightarrow C$:

- server returns its certificate chain of X.509v3

```
opaque ASN.1Cert<1..2^24-1>;
struct {
  ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;
```

Certificate Request

optional $S \rightarrow C$: server asks for client certificate

```
enum {
  rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
  rsa_ephemeral_dh(5), dss_ephemeral_dh(6), fortezza_kea(20),
  (255)
} ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;

struct {
  ClientCertificateType certificate_types<1..2^8-1>;
  DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;
```

Session Keys

1. 48-byte pre-master-secret s_p generated by client
2. master secret

$$\begin{aligned} s_m \quad = \quad & \text{MD5}(s_p | \text{SHA}('A' | s_p | r_c | r_s)) \\ & | \text{MD5}(s_p | \text{SHA}('BB' | s_p | r_c | r_s)) \\ & | \text{MD5}(s_p | \text{SHA}('CCC' | s_p | r_c | r_s)) \end{aligned}$$

   where $r_{c,s}$ client, server random

3. session key: same as above to generate byte stream
   cut out:
   - server, client MAC secret
   - server, client write key
   - server, client write IV

Client.hello random provide “salt”

Example: SSL for export

128 bit key, but only allow 40 random bits for RC2 key:

```
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
client_write_MAC_secret = key_block[0..15]
server_write_MAC_secret = key_block[16..31]
client_write_key      = key_block[32..36]
server_write_key      = key_block[37..41]
final_client_write_key = MD5(client_write_key +
                         ClientHello.random +
                         ServerHello.random)[0..15];
final_server_write_key = MD5(server_write_key +
                         ServerHello.random +
                         ClientHello.random)[0..15];
client_write_IV = MD5(ClientHello.random +
                  ServerHello.random)[0..7];
server_write_IV = MD5(ServerHello.random +
                  ClientHello.random)[0..7];
```

Client Key Exchange

- client verifies certificate chain against that in web browser
- if not in list of CAs, may trust the new certificate
- *client* (why?) generates pre-master key
- sends $\{\text{pre-master key}\}_{\text{server}}$:

$C \rightarrow S$:

```
struct {
  select (KeyExchangeAlgorithm) {
    case rsa: EncryptedPreMasterSecret;
    case diffie_hellman: ClientDiffieHellmanPublic;
    case fortezza_kea: FortezzaKeys;
  } exchange_keys;
} ClientKeyExchange;
struct {
  public-key-encrypted struct {
    ProtocolVersion client_version;
    opaque random[46];
  } PreMasterSecret;
} EncryptedPreMasterSecret;
```

Request Certificate

Optional, if desired: $S \rightarrow C$

- ask for certificate
- challenge phrase, encrypted with server-write key
- client responds: [MD5(server challenge and certificate), client certificate] $_{\text{client}}$
- server verifies certificate, hash (why?)
- $S \rightarrow C$: K_{sw} {session identifier} (why?)

Certificate Verify

If client sent certificate, $C \rightarrow S$:

```
struct {
  Signature signature;
} CertificateVerify;
```

$\text{MD5}(s_m|p_2|\text{MD5}(h|s_m|p_1))$

Finished

- decrypt master session key
- $S \rightarrow C$: K_{sw}

$C \rightarrow S, S \rightarrow C$:

```
enum { client(0x434C4E54), server(0x53525652) } Sender;
struct {
  opaque md5_hash[16];
  opaque sha_hash[20];
} Finished;
```

hash $= H(s_m|p_2|H(h|c|s_m|p_1))$
where H = MD5 or SHA; p_i: pad; h: handshake message; c: `Sender`

Use of Client Certificates

Netscape Enterprise server:

- same password table as for basic authentication
- sign up on first use to associate name, password with certificate
- only use certificate later

disadvantages of certificate: not portable