Symmetric Cryptography
+
Web Security

Tadayoshi Kohno

Thanks to Dan Boneh, Dieter Gollmann, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
International Criminal Tribunal for Rwanda


Credits: Alexei Czeskis, Karl Koscher, Batya Friedman
HMAC

- Construct MAC by applying a cryptographic hash function to message and key
- Invented by Bellare, Canetti, and Krawczyk (1996)
- Mandatory for IP security, also used in SSL/TLS
Structure of HMAC

- **Embedded hash function**
  - Strength of HMAC relies on the strength of this hash function.

- **"Black box"**: can use this HMAC construction with any hash function (why is this important?)

- **Block size of embedded hash function**

- **Secret key padded to block size**
  - "Amplify" key material (get two keys out of one)

- **Another magic value** (flips different key bits)

- **"Amplify" key material** (get two keys out of one)

- **Very common problem**: given a small secret, how to derive a lot of new keys?

- **Hash function**

- **Hash(key, hash(key, message))**
Achieving Both Privacy and Integrity

Authenticated encryption scheme

Recall: Often desire both privacy and integrity. (For SSH, SSL, IPsec, etc.)
Some subtleties! Encrypt-and-MAC

Natural approach for authenticated encryption: Combine an encryption scheme and a MAC.

\[ \bar{E}_{Ke,Km} \]

\[ E_{Ke,Km} \]

\[ \bar{D}_{Ke,Km} \]

\[ D_{Ke,Km} \]

Encrypt\(_{Ke}\) \rightarrow MAC\(_{Km}\) \rightarrow C' \rightarrow T

Ciphertext

Return \( M \) if valid

valid/invalid
But insecure! [BN, Kra]

Assume Alice sends messages:

If $T_i = T_j$ then $M_i = M_j$

Adversary learns whether two plaintexts are equal.

Especially problematic when $M_1, M_2, \ldots$ take on only a small number of possible values.
Results of [BN00, Kra01]

<table>
<thead>
<tr>
<th>Privacy</th>
<th>Encryption Method</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong (CCA)</td>
<td>Encrypt-then-MAC</td>
<td>Strong (CTXT)</td>
</tr>
<tr>
<td>Weak (CPA)</td>
<td>MAC-then-Encrypt</td>
<td>Weak (PTXT)</td>
</tr>
<tr>
<td>Insecure</td>
<td>Encrypt-and-MAC</td>
<td>Weak (PTXT)</td>
</tr>
</tbody>
</table>

- **MAC**: Message Authentication Code
- **Ke**: Encryption Key
- **M**: Original Message
- **C**: Ciphertext
- **T**: Tag or MAC
- **CTXT**: Ciphertext Integrity
- **CCA**: Chosen Ciphertext Attack
- **CPA**: Chosen Plaintext Attack
- **PTXT**: Plaintext Integrity
The Secure Shell (SSH) protocol is designed to provide:

- Secure remote logins.
- Secure file transfers.

Where security includes:

- Protecting the privacy of users’ data.
- Protecting the integrity of users’ data.

OpenSSH is included in the default installations of OS X and many Linux distributions.
Authenticated encryption in SSH

\[ E_{Ke,Km} \]

Data to be communicated

Maintained internally; not transmitted

\[
\begin{align*}
\text{ctr} & \quad \text{pl} \quad \text{pdl} \quad M \quad \text{padding} \\
4 \text{ bytes} & \quad 4 \text{ bytes} \quad 1 \text{ byte}
\end{align*}
\]

Encrypted packet

\[ C' \]

MAC

\[ MAC_{Km} \]

Ciphertext packet
What’s different about SSH?

Assume Alice sends messages $M_1$ and $M_2$ that are the same.

But if counters repeat, tags may once again leak private information about data.

Then the tags $T_1$ and $T_2$ will be different with high probability.
Now: Web Security
(Back to Asymmetric Cryptography Later)
Browser and Network

Browser

OS

Hardware

Network

website

request

reply
Types of problems

- **Web browser problems (client side)**
  - Exploit vulnerabilities in browsers
  - Install botnets, keyloggers
  - Exfiltrate data

- **Web application code (server side)**
  - Exploit vulnerabilities in code running on servers (and coming from servers)
  - Examples: XSS, XSRF, SQL injection, insecure parameters, security misconfigurations
  - Steal user credentials, data from databases, ...
Example Questions

- How does website know who you are?
- How do you know who the website is?
- Can someone intercept traffic?
- Related: How can you better control flow of information?

- Our focus: High-level principles (lab focuses on pragmatics)
- Focus on a bit of history: How we got here
HTTP: HyperText Transfer Protocol

- Used to request and return data
  - Methods: GET, POST, HEAD, ...

- Stateless request/response protocol
  - Each request is independent of previous requests
  - Statelessness has a significant impact on design and implementation of applications

- Evolution
  - HTTP 1.0: simple
  - HTTP 1.1: more complex
  - ... Still evolving ...
## HTTP Request

<table>
<thead>
<tr>
<th>Method</th>
<th>File</th>
<th>HTTP version</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/default.asp</td>
<td>HTTP/1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accept: image/gif, image/x-bitmap, image/jpeg, <em>/</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accept-Language: en</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Connection: Keep-Alive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If-Modified-Since: Sunday, 17-Apr-96 04:32:58 GMT</td>
</tr>
</tbody>
</table>

Data – none for GET

Blank line
HTTP Response

HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Content-Length: 2543

<HTML> Some data... blah, blah, blah </HTML>
Primitive Browser Session

View catalog
www.e_buy.com

Select item
www.e_buy.com/shopping.cfm?pID=269

Check out
www.e_buy.com/checkout.cfm?pID=269&item1=102030405

Store session information in URL; easily read on network
FatBrain.com circa 1999  [due to Fu et al.]

- User logs into website with his password, authenticator is generated, user is given special URL containing the authenticator

  https://www.fatbrain.com/HelpAccount.asp?t=0&p1=me@me.com&p2=540555758

- With special URL, user doesn’t need to re-authenticate
  - Reasoning: user could not have not known the special URL without authenticating first. That’s true, BUT...

- Authenticators are global sequence numbers
  - It’s easy to guess sequence number for another user

  https://www.fatbrain.com/HelpAccount.asp?t=0&p1=SomeoneElse&p2=540555752

- Partial fix: use random authenticators
Bad Idea: Encoding State in URL

- Unstable, frequently changing URLs
- Vulnerable to eavesdropping
- There is no guarantee that URL is private
  - Early versions of Opera used to send entire browsing history, including all visited URLs, to Google
Cookies
A **cookie** is a data blob created by an Internet site to store information on your computer.

HTTP is traditionally a stateless protocol; cookies add state.
What Are Cookies Used For?

- **Authentication**
  - Use the fact that the user authenticated correctly in the past to make future authentication quicker

- **Personalization**
  - Recognize the user from a previous visit

- **Tracking**
  - Follow the user from site to site; learn his/her browsing behavior, preferences, and so on
Cookie Management

◆ Cookie ownership
  - Once a cookie is saved on your computer, only the website that created the cookie can read it (supposedly)

◆ Variations
  - Temporary cookies
    - Stored until you quit your browser
  - Persistent cookies
    - Remain until deleted or expire
  - Third-party cookies
    - Set by sites embedded within other sites (e.g., ads)
Privacy Issues with Cookies

- Cookie may include any information about you known by the website that created it
  - Browsing activity, account information, etc.

- Sites can share this information
  - Advertising networks
  - 2o7.net tracking cookie

- Browser attacks could invade your privacy

  November 8, 2001 (and many more sense):
  Users of Microsoft's browser and e-mail programs could be vulnerable to having their browser cookies stolen or modified due to a new security bug in Internet Explorer (IE), the company warned today
Storing State in Browser

❄ Dansie Shopping Cart (2006)
  • “A premium, comprehensive, Perl shopping cart. Increase your web sales by making it easier for your web store customers to order.”

<FORM METHOD=POST
ACTION="http://www.dansie.net/cgi-bin/scripts/cart.pl">
  Black Leather purse with leather straps<br>Price: $20.00<br>
  <INPUT TYPE=HIDDEN NAME=name VALUE="Black leather purse">
  <INPUT TYPE=HIDDEN NAME=price VALUE="20.00">
  <INPUT TYPE=HIDDEN NAME=sh VALUE="1">
  <INPUT TYPE=HIDDEN NAME=img VALUE="purse.jpg">
  <INPUT TYPE=HIDDEN NAME=custom1 VALUE="Black leather purse with leather straps">
  <INPUT TYPE=SUBMIT NAME="add" VALUE="Put in Shopping Cart">
</FORM>

Change this to 2.00
Many Web-based shopping cart applications use hidden fields in HTML forms to hold parameters for items in an online store. These parameters can include the item's name, weight, quantity, product ID, and price. Any application that bases price on a hidden field in an HTML form is vulnerable to price changing by a remote user. A remote user can change the price of a particular item they intend to buy, by changing the value for the hidden HTML tag that specifies the price, to purchase products at any price they choose.

**Platforms Affected:**

- 3D3.COM Pty Ltd: ShopFactory 5.8 and earlier
- Adgrafx: Check It Out Any version
- ComCity Corporation: SalesCart Any version
- Dansie.net: Dansie Shopping Cart Any version
- Make-a-Store: Make-a-Store OrderPage Any version
- McMurtrey/Whitaker & Associates: Cart32 3.0
- McMurtry/Whitaker & Associates: Cart32 2.6
- McMurtrey/Whitaker & Associates: Cart32 2.6
- Rich Media Technologies: JustAddCommerce 5.0
- Web Express: Shoptron 1.2
- @Retail Corporation: @Retail Any version
- Baron Consulting Group: WebSite Tool Any version
- Crested Butte Software: EasyCart Any version
- Intelligent Vending Systems: Intellivend Any version
- McMurtrey/Whitaker & Associates: Cart32 2.6
- pknutsen@nethut.no: CartMan 1.04
- SmartCart: SmartCart Any version

[http://xforce.iss.net/xforce/xfdb/4621](http://xforce.iss.net/xforce/xfdb/4621)
Storing State in Browser Cookies

- Set-cookie: price=299.99
- User edits the cookie... cookie: price=29.99
- What’s the solution?
- Add a MAC to every cookie, computed with the server’s secret key
  - Price=299.99; MAC(ServerKey, 299.99)
- Is this the solution?
Storing State in Browser

◆ Dansie Shopping Cart (2006)
  - “A premium, comprehensive, Perl shopping cart. Increase your web sales by making it easier for your web store customers to order.”

  <FORM METHOD=POST
  ACTION="http://www.dansie.net/cgi-bin/scripts/cart.pl">
    Black Leather purse with leather straps<br>Price: $20.00<br>
    <INPUT TYPE=HIDDEN NAME=name VALUE="Black leather purse">
    <INPUT TYPE=HIDDEN NAME=price VALUE="F13A3....B2">
    <INPUT TYPE=HIDDEN NAME=sh VALUE="1">
    <INPUT TYPE=HIDDEN NAME=img VALUE="purse.jpg">
    <INPUT TYPE=HIDDEN NAME=custom1 VALUE="Black leather purse with leather straps">
    <INPUT TYPE=SUBMIT NAME="add" VALUE="Put in Shopping Cart">
  </FORM>

Better: MAC(K, “$20,Black leather purse, product number 12345, ...”)
Web Authentication via Cookies

- Need authentication system that works over HTTP and does not require servers to store session data

- Servers can use cookies to store state on client
  - When session starts, server computes an authenticator and gives it back to browser in the form of a cookie
    - Authenticator is a value that client cannot forge on his own
    - Example: MAC(server’s secret key, session id)
  - With each request, browser presents the cookie
  - Server recomputes and verifies the authenticator
    - Server does not need to remember the authenticator
Typical Session with Cookies

- **client**
  - POST /login.cgi
  - Set-Cookie: authenticator
  - GET /restricted.html
  - Cookie: authenticator
  - Restricted content

- **server**
  - Verify that this client is authorized
  - Check validity of authenticator (e.g., recompute hash(key, sessId))

Authenticators must be **unforgeable and tamper-proof** (malicious client shouldn’t be able to compute his own or modify an existing authenticator)
Idea: use `user,hash(user||key)` as authenticator

- Key is secret and known only to the server. Without the key, clients can’t forge authenticators.
- `||` is string concatenation

Implementation: `user,crypt(user||key)`

- `crypt()` is UNIX hash function for passwords
- `crypt()` truncates its input at 8 characters
- Usernames matching first 8 characters end up with the same authenticator
- No expiration or revocation

It gets worse... This scheme can be exploited to extract the server’s secret key
### Attack

<table>
<thead>
<tr>
<th>username</th>
<th>crypt(username,key,“00”)</th>
<th>authenticator cookie</th>
</tr>
</thead>
<tbody>
<tr>
<td>AliceBob1</td>
<td>008H8LRfzUXvk</td>
<td>AliceBob1008H8LRfzUXvk</td>
</tr>
<tr>
<td>AliceBob2</td>
<td>008H8LRfzUXvk</td>
<td>AliceBob2008H8LRfzUXvk</td>
</tr>
</tbody>
</table>

“Create” an account with a 7-letter user name...

AliceBoA 0073UYEre5rBQ Try logging in: access refused
AliceBoB 00bkHcfOXBXKno Access refused
AliceBoC 00ofSJV6An1QE Login successful! 1st key symbol is C

Now a 6-letter user name...

AliceBCA 001mBnBErXRuc Access refused
AliceBCB 00T3JLLfuspdo Access refused... and so on

- Only need 128 x 8 queries instead of intended 128^8
- Minutes with a simple Perl script vs. billions of years
Better Cookie Authenticator

Capability
Describes what user is authorized to do on the site that issued the cookie

Expiration

MAC(server secret, capability, expiration)
Cannot be forged by malicious user; does not leak server secret

◆Main lesson: be careful rolling your own
• Homebrewed authentication schemes are easy to get wrong
◆There are standard cookie-based schemes
Web Applications

◆ Online banking, shopping, government, etc.
◆ Website takes input from user, interacts with back-end databases and third parties, outputs results by generating an HTML page
◆ Often written from scratch in a mixture of PHP, Java, Perl, Python, C, ASP, ...
◆ Security is a potential concern.
  • Poorly written scripts with inadequate input validation
  • Sensitive data stored in world-readable files