Goals of Today’s Lecture

• Putting all that crypto goop together: *how does https work?*
  
• Attacks: compromising systems
  – Buffer overflows
  – Logic errors
  – Social engineering
  – Automated attacks: worms & bots
    ⇒ Leading to the rise of the underground economy

• Attacks: denial-of-service (DoS)
  – Network layer (DDoS)
  – Transport layer (SYN flooding)
  – Application layer (bogus/expensive requests)
Putting It All Together - HTTPS

- What happens when you click on https://www.amazon.com?

- **https** = “Use HTTP over SSL/TLS”
  - **SSL** = Secure Socket Layer
  - **TLS** = Transport Layer Security
    - Successor to SSL, and compatible with it
    - RFC 4346
    - Provides security layer (authentication, encryption) on top of TCP
    - Fairly transparent to the app

HTTPS Connection (SSL/TLS), con’t

- Browser (client) connects via TCP to Amazon’s HTTPS server
- Client sends over list of crypto protocols it supports
- Server picks protocols to use for this session
- Server sends over its certificate
- (all of this is in the clear)
Inside the Server’s Certificate

- **Name** associated with cert (e.g., Amazon)
- Amazon’s **public key** *(RSA exponent e, modulus n)*
- A bunch of auxiliary info (physical address, type of cert, expiration time)
- URL to *revocation center* to check for revoked keys
- Name of certificate’s **signatory** (who signed it)
- A public-key signature of a hash *(MD5)* of all this
  – Constructed using the signatory’s private RSA key

Validating Amazon’s Identity

- Browser retrieves cert belonging to the **signatory**
  – These are **hardwired into the browser**
- If it can’t find the cert, then warns the user that site has not been verified
  – And may ask whether to continue
  – Note, can still proceed, just **without authentication**
- Q: have you ever told your browser to proceed in this situation? What’s the big deal?
The Perils of Ignoring SSL Warnings

whois -h whois.ripe.net 91.203.92.63
... address: Ukraine, Vornesensk, Lenina 52

Validating Amazon’s Identity

• Browser retrieves cert belonging to the **signatory**
  – These are **hardwired into the browser**

• If it can’t find the cert, then warns the user that site has not been verified
  – And may ask whether to continue
  – Note, can still proceed, just **without authentication**

• Browser uses public key in signatory’s cert to decrypt signature
  – Compares with its own **MD5** hash of Amazon’s cert

• Assuming signature matches, now have high confidence it’s indeed Amazon ...
  – ... **assuming signatory is trustworthy**
HTTPS Connection (SSL/TLS), con’t

- Browser constructs a random session key $K$
- Browser encrypts $K$ using Amazon’s public key
- Browser sends $E(K, \{n, e\})$ to server
- Browser displays
  - All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) using key $K$
    - E.g., client can authenticate using a password
  - (what step is missing?)

Host Compromise

- **Tricking** a host into executing on your behalf
- Can consider *what* is attacked (server or client) and the *semantic level* at which it is attacked
- Attacks on servers: client sends subversive requests
  - Happens at attacker’s choosing
  - *Some hosts are servers unknowingly!*
- Attacks on clients: server (attacker) *waits* for client to connect, sends it a subversive reply
  - E.g., “drive-by” spyware / web infections
  - E.g., 2006 study found 15% of popular P2P files infected by one of 52 different viruses
Semantic Level of Compromise

• **E.g., buffer overflows**
  – Part of the request sent by the attacker **too large** to fit into buffer server uses to hold it.
  – Spills over into memory beyond the buffer
  – Allows **remote** attacker to **inject** executable code

• **Large** class of attacks, with a variety of defenses
  – (Host-based: randomized layouts, detection of overwritten memory, execution of network payload, impossible call stacks)
  – (Network-based: signatures, semantic analysis, post-attack activity)

• **Violates semantics of underlying programming language**

---

Example

```c
void get_cookie(char *packet) {
    // (200 bytes of local vars)
    munch(packet);

    // code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);

    // ...
}

void munch(char *packet) {
    int n;
    char cookie[512];

    // ...
}
```
Example: Normal Execution

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Example: Normal Execution

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```
Example: Normal Execution

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in
    packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Stack

```
get_cookie()’s
stack frame
```

return address back
to get_cookie()

```
X + 200
```

```
X
```

```
X - 4
```

```
X - 8
```

```
X - 520
```

cookie

Example: Normal Execution

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in
    packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Stack

```
get_cookie()’s
stack frame
```

return address back
to get_cookie()

```
X + 200
```

```
X
```

```
X - 4
```

```
X - 8
```

```
X - 520
```
Example: Normal Execution

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}
void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    // code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Stack

- `get_cookie()`'s stack frame
- Return address back to `get_cookie()`
- `n`
- `cookie`
- Return address back to `munch()`
- `strcpy()`'s stack ...
Example: Normal Execution

```c
void get_cookie(char *packet) {
    ... (200 bytes of local vars) ...
    munch(packet);
    ...
}

void munch(char *packet) {
    int n;
    char cookie[512];
    ...
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    ...
}
```

Stack diagram:
- `get_cookie()`'s stack frame
- Return address back to `get_cookie()`
- `n`
- Cookie value read from packet
- Stack:
  - `get_cookie()`'s stack frame
  - Return address back to `get_cookie()`
  - `n`
  - Cookie value read from packet

Example: Normal Execution

```c
void get_cookie(char *packet) {
    ... (200 bytes of local vars) ...
    munch(packet);
    ...
}

void munch(char *packet) {
    int n;
    char cookie[512];
    ...
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    ...
}
```
Example: Normal Execution

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
}
void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Example: Buffer Overflow

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
}
void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```
Example: Buffer Overflow

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Executable Code

Stack

Example: Buffer Overflow

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```
Example: Buffer Overflow

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Example: Buffer Overflow

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}

void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
Example: Buffer Overflow

```c
void get_cookie(char *packet) {
    . . . (200 bytes of local vars) . . .
    munch(packet);
    . . .
}
void munch(char *packet) {
    int n;
    char cookie[512];
    . . .
    . . .
    code here computes offset of cookie in packet, stores it in n
    strcpy(cookie, &packet[n]);
    . . .
}
```

Now branches to code read in from the network

From here on, machine falls under the attacker's control

5 Minute Break

Questions Before We Proceed?
Buffer Overflows: Potential Solutions

• Don’t write buggy software
  – It’s not like people try to write buggy software

• Type-safe Languages
  – Unrestricted memory access of C/C++ contributes to problem
  – Use Java, Perl, Python instead

• OS architecture
  – Compartmentalize programs better, so one compromise doesn’t compromise the entire system
  – E.g., DNS server doesn’t need total system access

• Firewalls - restrict remote access to services
• Intrusion detection: recognize attack & block it

Semantic Level of Compromise, con’t

• Logic errors

• E.g., suppose your Web server passes any argument named “rev” in a URL request to a backend script called munch via the equivalent of
  
  sh munch $rev

  and returns its output

• Now suppose you receive the following request:

  GET /bin/TWikiUsers?rev=2%20|cat%20/etc/passwd

  It decodes to:

  $rev = “2 |cat /etc/passwd”
Logic Errors, con’t

• Your script is invoked as
  ```bash
  sh munch 2 | cat /etc/passwd
  ```
  which returns as output the password file.

• Similar “SQL injection” attacks acquire access / modify backend databases

• Note: no violation of programming semantics!
  ⇒ Very hard to detect. Need to understand intended semantics.

• Similar problems occur any time executable content is allowed
  • E.g., Web plug-ins, document macros

Semantic Level of Compromise, con’t

• Social engineering: misleading/fooling humans

• E.g., DNS typo attacks (register `www.googole.com`)

• E.g., phishing

• E.g., “You Have A Greeting Card!”
Welcome to Storm!

Would you like to be one of our newest bots?
Just read your postcard!
(Or even easier: just wait 5 seconds!)

Semantic Level of Compromise, con’t

- Social engineering: misleading/fooling humans
- E.g., DNS typo attacks (register www.gooogle.com)
- E.g., phishing
- E.g., “You Have A Greeting Card!”
- Powerful technique for targeted attacks
  - E.g., find out name & mailstop of a company’s sysadmin
  - … mail an employee bogus system CD as if from them
    - With a note that it contains an important security update
  - User trusts source of update, applies it
    ⇒ They install a backdoor into company
- General defense: user education :-(

©2000-2008 AwesomePostCard.com - All rights reserved.
Automated Compromise: Worms

- When attacker compromises a host, they can instruct it to do whatever they want
- Instructing it to find more vulnerable hosts to repeat the process creates a **worm**: a program that self-replicates across a network
  - Often spread by picking 32-bit Internet addresses at random to probe …
  - … but this isn’t fundamental
- As the worm repeatedly replicates, it grows **exponentially fast** because each copy of the worm works in parallel to find more victims

Worms: Exponentially Fast …. and Big

- Code Red 1 (2001)
  - 369K hosts in 10 hours
- Blaster (2003)
  - 9M hosts in 9 days
  - 25M hosts total
- Slammer (2003)
  - 75K hosts …
  - … in < 10 minutes
  - Peak scanning rate:
    - 55M addresses/sec
    - *Limited by Internet’s capacity*
- Theoretical worms
  - 1M hosts in 1.3 sec (2004)
Automated Compromise: Bots

- Big worms are flashy but rare …
- … With the commercialization of malware, the tool of choice has shifted to the less noisy, more directly controlled botnets
- When host is (automatically) compromised, don’t continue propagation
  - Instead install a command and control platform (a bot)
- Now can monetize malware: sell access to bots
- Spamming, phishing web sites, flooding attacks
- “Crook’s Google Desktop”: sell capability of searching the contents of 100,000s of hosts
- (Note: we still worry about worms for cyberwarfare)
79. ANCHETA would accept payments through Paypal.

103. In or about August 2004, ANCHETA updated his advertisement to increase the price of bots and proxies, to limit the purchase of bots to 2,000 “due to massive orders,” and to warn,

adware on those computers without notice to or consent from the users of those computers, and by means of such conduct, obtained the following approximate monies from the following advertising service companies:

<table>
<thead>
<tr>
<th>COUNT</th>
<th>APPROXIMATE DATES</th>
<th>APPROXIMATE NUMBER OF PROTECTED COMPUTERS ACCESSED WITHOUT AUTHORIZATION</th>
<th>APPROXIMATE PAYMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEVEN</td>
<td>November 1, 2004 through November 19, 2004</td>
<td>26,975</td>
<td>$4,044.26 from Gammacash</td>
</tr>
<tr>
<td>EIGHT</td>
<td>November 16, 2004 through December 7, 2004</td>
<td>8,744</td>
<td>$1,306.52 from LOODcash</td>
</tr>
<tr>
<td>NINE</td>
<td>January 15, 2005</td>
<td>10,434</td>
<td>$2,500.00</td>
</tr>
</tbody>
</table>
Spy Instructors Software
NEW GENERATION SOFTWARE SOLUTIONS

- ProAgent Spy Software is one of the most powerful monitoring and surveillance applications available today.
- It is an atomic solution for monitoring spouses, children, employees, or anyone else.
- ProAgent records all typed keywords, all active windows tasks, all visited web sites, all instant messages, passwords and more, and sends e-mail reports to your e-mail address that you specified when creating the server, completely hidden.
- ProAgent can work in all kind of networks, it doesn't matter if the PC is behind a firewall or behind a router or in a LAN.
- ProAgent works in all of these conditions without any problems.

Click here to purchase ProAgent v2.1 Special Edition...
Click here to download ProAgent v2.1 Basic Edition.

SIS - Products
- Purchased Products
- Customer Support Department
- Commercial Programs
- Freeware Programs
- Custom Special Programs

New Products
- SIS-Explorer v2.0
- ProAgent v2.1

Andres v1.2
SIS-Downloader
Virtual Keyboard

---

Список доступных аккаунтов

Добрый вечер! Представляем Вашему вниманию свои аккаунты eBay.

Постоянным клиентам и тем, кто берет более 5 аккаунтов, различные бонусы и скидки.

Все аккаунты с доставкой на почтовый адрес.

Вы сами выбираете аккаунт (несколько аккаунтов) из списка. Говорите мне, какой вы хотите.

Все аккаунты проверены перед продажей, в случае, если что-то не работает, 100% возврат.

Активные аккаунты с более 10 фидов. По активности не сортирую, так как это для каждого индивидуально.

Также в продаже бывают аккаунты PayPal. Цены разные. Постоянно не прекращаю.

Оплата по WM, на карту, через PayPal.

Просмотр аккаунтов осуществляется через FAQ.

По работе с товаром не консультирую.

Работа через гарантийный сервис не проводится.

Условия:

seller/baer аккаунт до 10 фидов - 55
seller/baer аккаунт 10-25 фидов - 105
seller/baer аккаунт 25-50 фидов - 155
seller/baer аккаунт более 50 фидов - 255
allBots Inc.
Social Networking Bots
GOOD News!!! We have something more for you! Yes, we have an integrated CAPTCHA Bypass in all of our bots.

Winsock (Multi-threaded) Bots
Click here for 30+ MySpace Bots

Become an Affiliate and Start Earning Now

### Accounts Creator
(You Just Need To Type In The CAPTCHAs To Create Accounts)

<table>
<thead>
<tr>
<th>Social Networks</th>
<th>MySpace Accounts Creator with Picture Uploader, Profile &amp; Layout Manager</th>
<th>$180.00</th>
<th>$140.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySpace Accounts Creator with Picture Uploader, Profile &amp; Layout Manager (Winsock)</td>
<td>$160.00</td>
<td>$20.00</td>
<td></td>
</tr>
<tr>
<td>YouTube Accounts Creator</td>
<td>$20.00</td>
<td>$95.00</td>
<td></td>
</tr>
<tr>
<td>Friendster Accounts Creator</td>
<td>$20.00</td>
<td>$95.00</td>
<td></td>
</tr>
<tr>
<td>Hi5 Accounts Creator</td>
<td>$20.00</td>
<td>$95.00</td>
<td></td>
</tr>
<tr>
<td>TopWorld Accounts Creator</td>
<td>PayPal</td>
<td>PayPal</td>
<td></td>
</tr>
</tbody>
</table>

### Friend Adders, Message Senders, Comment Posters & Others
(All Bots Work In A Conventional Manner, They Gather Friend IDs/Names And Send Friend Requests, Messages, Comments Automatically)

**Email Feeding** is Available On All Bots for All Networks except Facebook

---

### Marketplace Ads for Goods

![Graph showing percentage of labeled data for different ad types](image)
Marketplace Ads for Services

Denial of Service (DoS)

- Attacker prevents legitimate users from using something (network, server)
- Motives?
  - Retaliation
  - Extortion (e.g., betting sites just before big matches)
  - Commercial advantage (disable your competitor)
  - Cripple defenses (e.g., firewall) to enable broader attack
  - Political statement / military advantage
- Often done via some form of **flooding**
- Can be done at different semantic levels
  - Network: clog link or router w/ a huge rate of packets
  - Transport: overwhelm victim’s handling of connections
  - Application: overwhelm victim’s handling of requests
DoS: *Network Flooding*

- Goal is to clog network link(s) leading to victim
  - Either fill the link, or overwhelm their routers
  - Users can’t access victim server due to congestion

- Attacker sends traffic to victim as fast as possible
  - It will often use (many) spoofed source addresses …

- Using multiple hosts (*bots*, or *slaves*) yields a *Distributed Denial-of-Service* attack, aka **DDoS**

- Traffic is *varied* (sources, destinations, ports, length) so no simple filter matches it

- If attacker has enough slaves, *often doesn’t need to spoof* - victim can’t shut them down anyway! :-(

---

**Distributed Denial-of-Service (DDoS)**

- *Control traffic directs slaves at victim*
- *Slaves send streams of traffic (perhaps spoofed) to victim*

- *src = random\ndst = victim*
Very Nasty DoS Attack: Reflectors

- **Reflection**
  - Cause one *non-compromised* host to help flood another
  - E.g., host A sends DNS request or TCP SYN with source V to server R.

![Diagram of Very Nasty DoS Attack: Reflectors]

**Diagram Description:**
- **Attacker (A)** sends a request to **Server R**.
- **Server R** reflects the request to **Victim (V)**.
- **Internet** facilitates the communication between the hosts.

Very Nasty DoS Attack: Reflectors

- **Reflection**
  - Cause one *non-compromised* host to attack another
  - E.g., host A sends DNS request or TCP SYN with source V to server R.
  - R sends reply to V

![Diagram of Very Nasty DoS Attack: Reflectors]

**Diagram Description:**
- **Attacker (A)** sends a request to **Server R**.
- **Server R** holds the request for a period of time.
- **Server R** reflects the request to **Victim (V)**.
- **Internet** facilitates the communication between the hosts.
**Defend DDoS: Reflector Attack**

Request: src = victim
dst = reflector

Reply: src = reflector
dst = victim

Master
Slave 1
Slave 2
Slave 3
Slave 4
Reflector 1
Reflector 2
Reflector 3
Reflector 4
Reflector 5
Reflector 6
Reflector 7
Reflector 8
Reflector 9
Reflector 10
Reflector 11

Control traffic directs slaves at victim & reflectors

Reflectors send streams of non-spoofed but unsolicited traffic to victim

---

**Defending Against Network Flooding**

- How do we defend against such floods?
- **Answer**: basically, we don’t! **Big** problem today!
- Techniques exist to **trace spoofed traffic** back to origins, but this isn’t useful in face of a large attack
- Techniques exist to **filter traffic**, but a well-designed flooding stream defies stateless filtering
- **Best solutions to date:**
  - **Overprovision** - have enough raw capacity that it’s hard to flood your links
    - Largest confirmed botnet to date: 1.5 million hosts
    - Floods seen to date: 40+ Gbps
  - **Distribute** your services - force attacker to flood many points
    - E.g., the root name servers
Summary

• The Web’s security rests on HTTPS
  – Implemented using SSL/TLS
  – Uses PKI certificates, public-key crypto, secret-key crypto, cryptographically strong hashing

• Attacks that compromise a system can occur at different semantic levels
  – E.g., Buffer overflow vs. cross-site-scripting vs. social engineering
  – Automated attacks lead to worms and bots

• The ready compromise of huge numbers of bots has led to the rise of a vibrant underground economy
  – Bodes ill for the ongoing “arms race” in Internet attacks/defenses

• Denial-of-service via flooding likewise can occur at different semantic levels
  – Network layer vs. transport layer vs. application layer
  – Very hard to address if attacker has a lot of zombies