CS162
Operating Systems and Systems Programming
Lecture 22

Security (II)

November 25, 2013

Anthony D. Joseph and John Canny

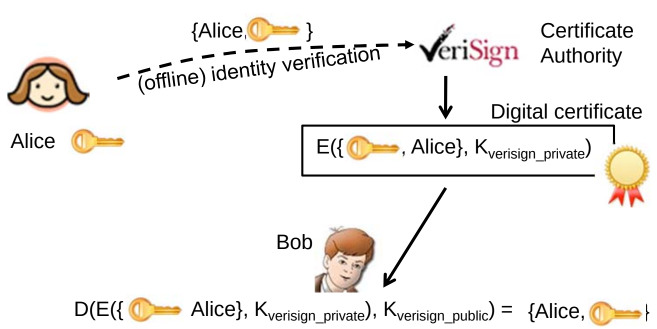
http://inst.eecs.berkeley.edu/~cs162

Recap: Security Requirements in Distributed Systems

- Authentication
  - Ensures that a user is who is claiming to be
- Data integrity
  - Ensure that data is not changed from source to destination or after being written on a storage device
- Confidentiality
  - Ensures that data is read only by authorized users
- Non-repudiation
  - Sender/client can't later claim didn't send/write data
  - Receiver/server can't claim didn't receive/write data

Recap: Digital Certificates

- How do you know [key] is Alice's public key?
- Main idea: trusted authority signs a binding (Alice's public key, Alice) with its private key.



Goals for Today

- Host Compromise
  - Attacker gains control of a host
- Denial-of-Service
  - Attacker prevents legitimate users from gaining service
- Attack can be both
  - E.g., host compromise that provides resources for denial-of-service

Host Compromise

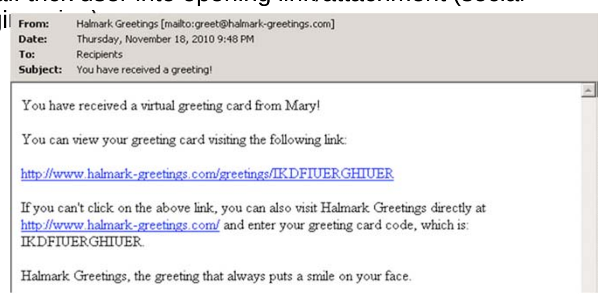
- One of earliest major Internet security incidents
  – Morris Worm (1988): compromised almost every BSD-derived machine on Internet
- Today: estimated that a single worm could compromise 10M hosts in < 5 min using a zero-day exploit
- Attacker gains control of a host
  – Reads data
  – Compromises another host
  – Launches denial-of-service attack on another host
  – Erases data

Definitions

- Worm
  – Replicates itself usually using buffer overflow attack
- Virus
  – Program that attaches itself to another (usually trusted) program or document
- Trojan horse
  – Program that allows a hacker a back door to compromised machine
- Botnet (Zombies)
  – A collection of programs running autonomously and controlled remotely
  – Can be used to spread out worms, mounting DDoS attacks

Trojan Example

- Nov/Dec e-mail message sent containing holiday message and a link or attachment
- Goal: trick user into opening link/attachment (social engi


From: Halmark Greetings [mailto:greet@halmark-greetings.com]
Date: Thursday, November 18, 2010 9:48 PM
To: Recipients
Subject: You have received a greeting!

You have received a virtual greeting card from Mary!

You can view your greeting card visiting the following link:

http://www.halmark-greetings.com/greetings/IKDFIUERGHIUER

If you can't click on the above link, you can also visit Halmark Greetings directly at http://www.halmark-greetings.com/ and enter your greeting card code, which is: IKDFIUERGHIUER.

Halmark Greetings, the greeting that always puts a smile on your face.

- Adds keystroke logger or turns into zombie
- How? Typically by using a buffer overflow exploit

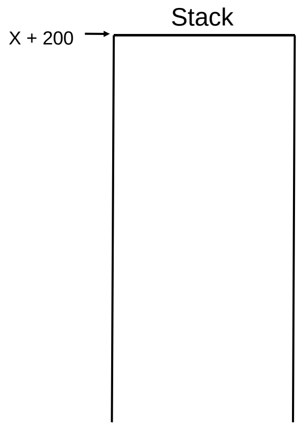
Buffer Overflow

- Part of the request sent by the attacker too large to fit into buffer program uses to hold it
- Spills over into memory beyond the buffer
- Allows remote attacker to inject executable code

```
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]);
  . . .
}
```

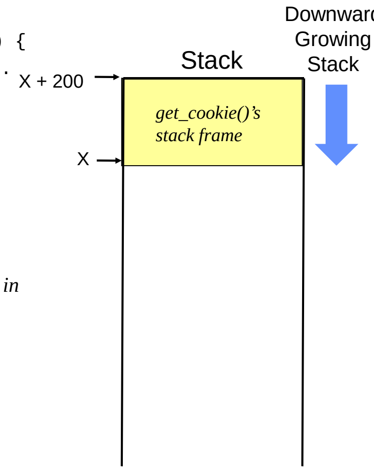
Example: Normal Execution

```
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]);
  . . .
}
```



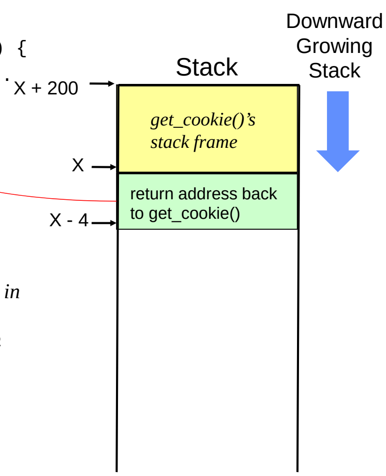
Example: Normal Execution

```
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]);
  . . .
}
```



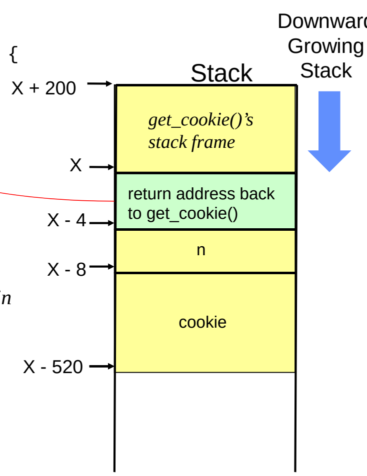
Example: Normal Execution

```
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]);
  . . .
}
```



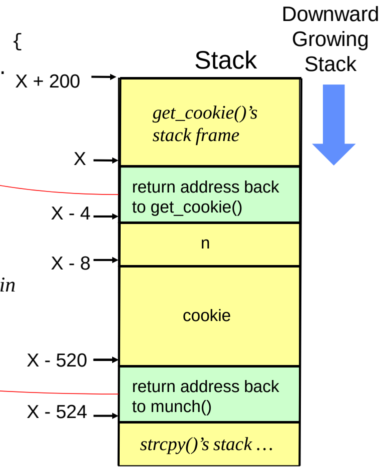
Example: Normal Execution

```
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]);
  . . .
}
```



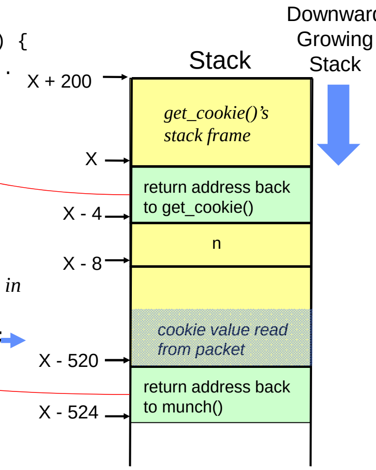
Example: Normal Execution

```
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]);
  . . .
}
```



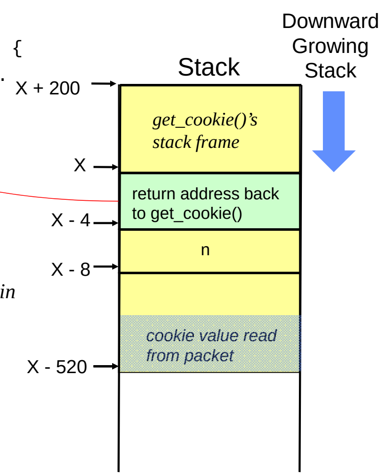
Example: Normal Execution

```
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]);
  . . .
}
```



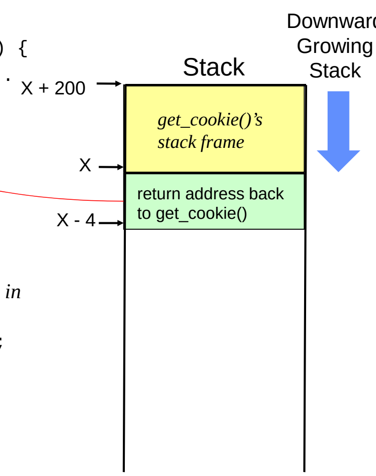
Example: Normal Execution

```
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]);
  . . .
}
```



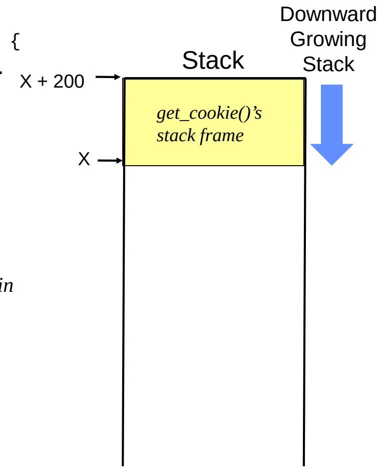
Example: Normal Execution

```
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]);
  . . .
}
```



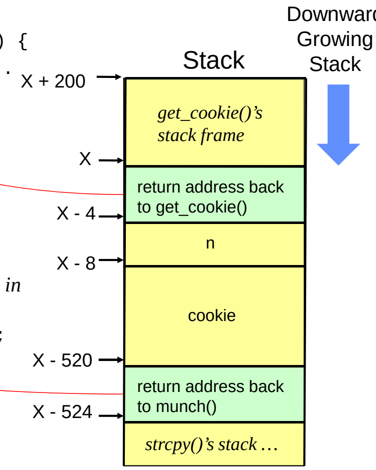
Example: Normal Execution

```
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]);
  . . .
}
```



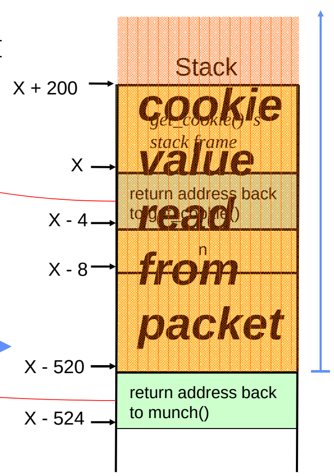
Example: Buffer Overflow

```
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]);
  . . .
}
```



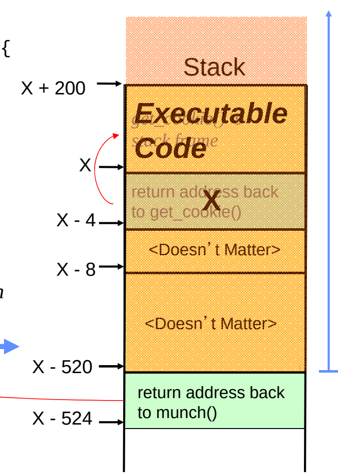
Example: Buffer Overflow

```
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]);
  . . .
}
```



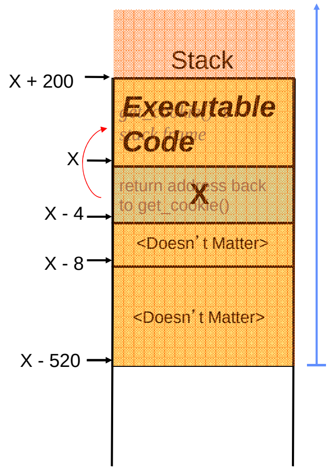
Example: Buffer Overflow

```
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

```
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]);
  . . .
}
```

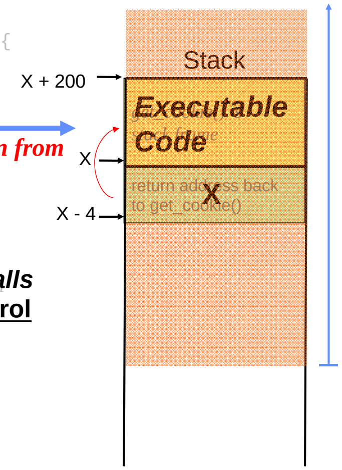


Example: Buffer Overflow

```
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



Buffer Overflow

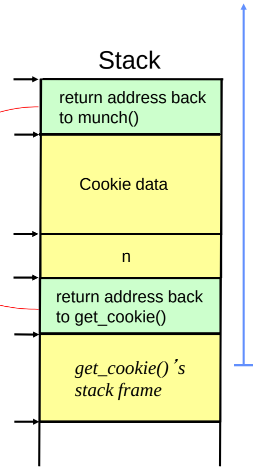
- The scenario above depended on the stack growing down.
- Can we prevent these kinds of overruns by growing the stack up instead – so overruns run into empty space instead of the stack?

Buffer Overflow

- The scenario above depended on the stack growing down.
- Can we prevent these kinds of overruns by growing the stack up instead – so overruns run into empty space instead of the stack?
- Not very effective – there are other opportunities to write into a return address.

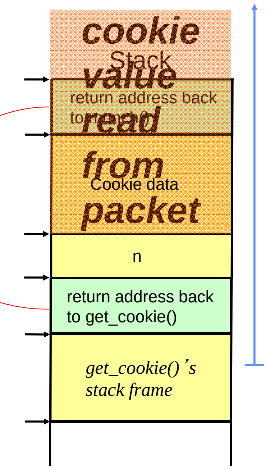
Buffer Overflow in upward stack

```
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

```
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]);
  . . .
}
```



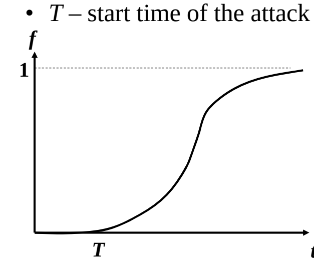
Automated Compromise: Worms

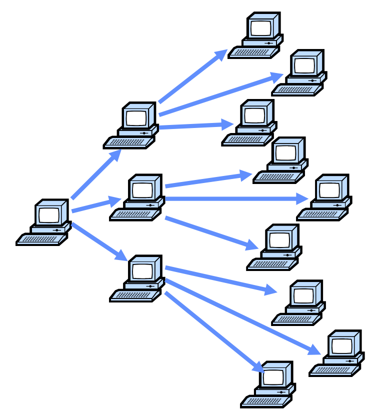
- When attacker compromises a host, they can instruct it to do whatever they want
- Instructing it to find more vulnerable hosts to repeat the process creates a worm: a program that self-replicates across a network
  - Often spread by picking 32-bit Internet addresses at random to probe …
  - … but this 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

Worm Spreading

$$f = (e^{K(t-T)} - 1) / (1 + e^{K(t-T)})$$

- f – fraction of hosts infected
- K – rate at which one host can compromise others
- T – start time of the attack





Worm Examples

- Morris worm (1988)
- Code Red v2 (2001)
  - 369K hosts in 10 hours
- MS Slammer (January 2003)
  - Around 70k hosts in 10 minutes
- Theoretical worms
  - Zero-day exploit, efficient infection and propagation
  - 1M hosts in 1.3 sec
  - $50B+ damage

Morris Worm (1988)

- Infect multiple types of machines (Sun 3 and VAX)
  - Was supposed to be benign: estimate size of Internet
- Used multiple security holes including
  - Buffer overflow in `fingerd`
  - Debugging routines in `sendmail`
  - Password cracking
- Intend to be benign but it had a bug
  - Fixed chance the worm wouldn't quit when reinfecting a machine → number of worm on a host built up rendering the machine unusable

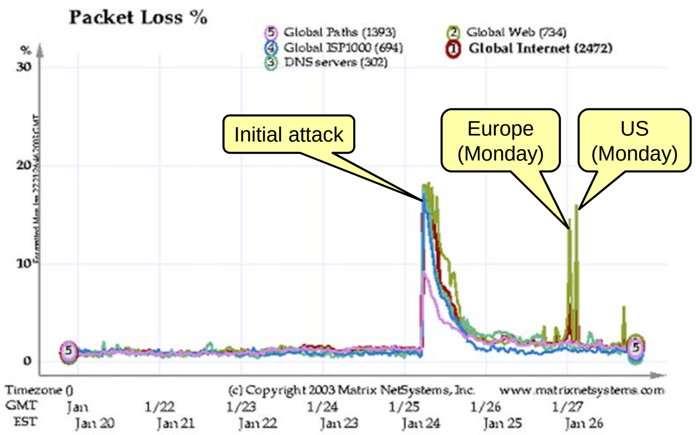
Code Red Worm (2001)

- Attempts to connect to TCP port 80 (i.e., HTTP port) on a randomly chosen host
- If successful, the attacking host sends a crafted HTTP GET request to the victim, attempting to exploit a buffer overflow
- Worm "bug": all copies of the worm use the same random generator and seed to scan new hosts
  - DoS attack on those hosts
  - Slow to infect new hosts
- 2nd generation of Code Red fixed the bug!
  - It spread much faster

MS SQL Slammer (January 2003)

- Host zero never found
- Author never found
- Average programmer
  - several bugs in random number generator
  - significant chunks of IPV4 address space not covered and therefore safe.

MS SQL Slammer (January 2003)



(From http://www.f-secure.com/v-descs/mssqlm.shtml)

MS SQL Slammer (January 2003)

- Uses UDP port 1434 to exploit a buffer overflow in MS SQL server
  - 376-bytes plus UDP and IP headers: one packet
- Effect
  - Generate massive amounts of network packets
  - Brought down as many as 5 of the 13 internet root name servers
- Others
  - The worm only spreads as an in-memory process: it never writes itself to the hard drive
    - Solution: close UDP port on firewall and reboot

Hall of Shame

- Software that have had many stack overflow bugs:
  - BIND (most popular DNS server)
  - RPC (Remote Procedure Call, used for NFS)
    - NFS (Network File System), widely used at UCB
  - Sendmail (most popular UNIX mail delivery software)
  - IIS (Windows web server)
  - SNMP (Simple Network Management Protocol, used to manage routers and other network devices)

Potential Solutions

- Don't write buggy software
  - Program defensively – validate all user-provided inputs
  - Use code checkers (slow, incomplete coverage)
- Use Type-safe Languages (Java, Perl, Python, …)
  - Eliminate unrestricted memory access of C/C++
- Use HW support for no-execute regions (stack, heap)
- Leverage OS architecture features
  - Address space randomization – randomize memory layout
  - Compartmentalize programs
    - E.g., DNS server doesn't need total system access
- Add network firewalls

Administrivia

- MIDTERM II 5:30-7pm in 145 Dwinelle (A-L) and 2060 Valley LSB (M-Z)
  - Review: TBA
  - Covers Lectures #14-24, projects, and readings
  - One sheet of notes, both sides
- Should be working on Project 4
  - Last one!
  - Initial Design Due **Monday**

5min Break

Quiz 22.1: Security

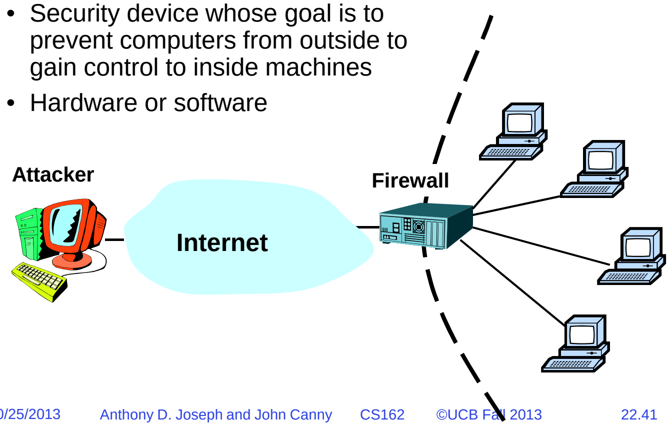
- Q1: True _ False _ A digital certificate provides a binding between a host's identity and their public key
- Q2: True _ False _ A server must store a user's password in plaintext form so it can be checked against a submitted password
- Q3: True _ False _ Worms require human intervention to propagate
- Q4: True _ False _ Using a type-safe language eliminates the risk of buffer overflows

Quiz 22.1: Security

- Q1: True X False _ A digital certificate provides a binding between a host's identity and their public key
- Q2: True _ False X A server must store a user's password in plaintext form so it can be checked against a submitted password
- Q4: True _ False X Worms require human intervention to propagate
- Q5: True X False _ Using a type-safe language eliminates the risk of buffer overflows

Firewall

- Security device whose goal is to prevent computers from outside to gain control to inside machines
- Hardware or software



Firewall (cont'd)

- Restrict traffic between Internet and devices (machines) behind it based on
  - Source address and port number
  - Payload
  - Stateful analysis of data
- Examples of rules
  - Block any external packets not for port 80 (i.e., HTTP port)
  - Block any email with an attachment
  - Block any external packets with an internal IP address
    - » Ingress filtering

Firewalls: Properties

- Easier to deploy firewall than secure all internal hosts
- Doesn't prevent user exploitation/social networking attacks
- Tradeoff between availability of services (firewall passes more ports on more machines) and security
  - If firewall is too restrictive, users will find way around it, thus compromising security
  - E.g., tunnel all services using port 80

Denial of Service

- Huge problem in current Internet
  - Major sites attacked: Yahoo!, Amazon, eBay, CNN, Microsoft
  - 12,000 attacks on 2,000 domains in 1 week (2001)
  - Almost all attacks launched from compromised hosts
- CyberBunker.com 300Gb/s DDoS attack against Spamhaus
  - Spring 2013: more than 600,000 packets/second!
  - 35 yr old Dutchman "S.K." arrested in Spain on 4/26
  - Was using van with "various antennas" as mobile office
- General Form
  - Prevent legitimate users from gaining service by overloading or crashing a server
  - E.g., SYN attack

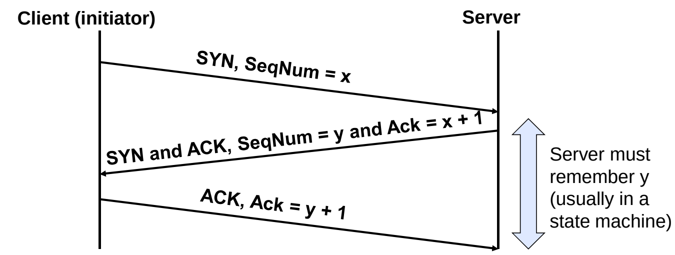
Effect on Victim

- Buggy implementations allow unfinished connections to eat all memory, leading to crash
- Better implementations limit the number of unfinished connections
  - Once limit reached, new SYNs are dropped
- Effect on victim's users
  - Users can't access the targeted service on the victim because the unfinished connection queue is full → DoS

SYN Attack

(Recap: TCP 3-Way Handshaking)

- Goal: agree on a set of parameters: the start sequence number for each side
  - Starting sequence numbers are random.



SYN Attack

- Attacker: send at max rate TCP SYN with random spoofed source address to victim
  - Spoofing: use a different source IP address than own
  - Random spoofing allows one host to pretend to be many
- Victim receives many SYN packets
  - Send SYN+ACK back to spoofed IP addresses
  - Holds some memory until 3-way handshake completes
    - » Usually never, so victim times out after long period (e.g., 3 minutes)

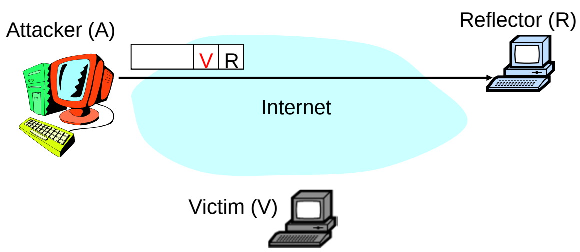
Solution: SYN Cookies

- Server: send SYN-ACK with sequence number y, where
  - y = HMAC(client_IP_addr, client_port, server_key)
  - HMAC(): Hash Message Authentication Code

  and forget about the connection attempt (don't use any resources).
- Client: send ACK containing y+1
- Server:
  - verify if y = HMAC(client_IP_addr, client_port, server_key)
  - If verification passes, allocate memory
- Note: server doesn't allocate any memory if the client's address is spoofed

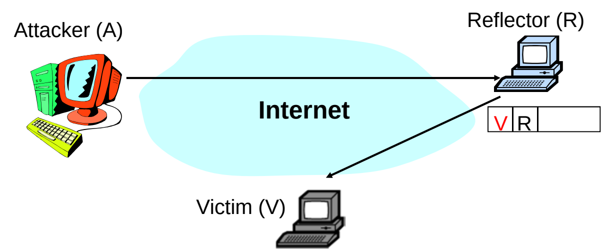
Other Denial-of-Service Attacks

- 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



Other Denial-of-Service Attacks

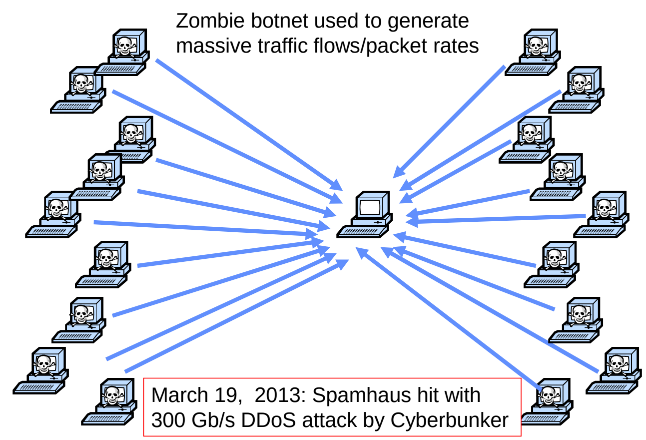
- Reflection
  - Cause one non-compromised host to attack another
  - E.g., host A sends DNS request or TCP SYN with source V to server R. R sends reply to V



Identifying and Stop Attacking Machines

- Develop techniques for defeating spoofed source addresses
- Egress filtering
  - A domain's border router drop outgoing packets which do not have a valid source address for that domain
  - If universal, could abolish spoofing
- IP Traceback
  - Routers probabilistically tag packets with an identifier
  - Destination can infer path to true source after receiving enough packets

Distributed Denial-of-Service Attacks



Two-Factor Authentication

- Authentication typically involves:
  - Something the user knows (e.g. password, friend's face)
  - Something the user has (ATM card, fob, dongle)
  - Something the user is (face, voice, fingerprints, bio-signs)
- Two-factor authentication involves two of these factors

Stepping Stone Compromise

- Today's most sophisticated attacks
  - Multi-step/compromise attack
- RSA SecurID token
  - 2-factor authentication device
  - Code changes every few seconds
  - *Data on codes stolen in March 2011*
- 760 companies attacked using stolen SecurID info
  - 20% of Fortune 100
  - Charles Schwabb & Co., Cisco Systems, eBay, European Space Agency, Facebook, Freddie Mac, Google, General Services Administration, IBM, Intel Corp., IRS, MIT, Motorola, Northrop Grumman, Verisign, VMWare, Wachovia, Wells Fargo, …
  - http://krebsonsecurity.com/2011/10/who-else-was-hit-by-the-rsa-attackers/

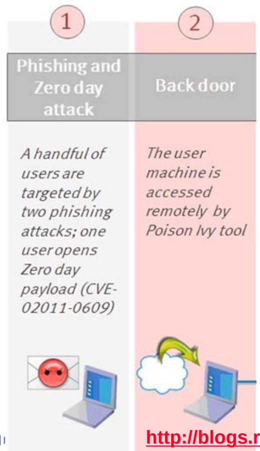
Advanced Persistent Threats

 http://blogs.rsa.com/rivner/anatomy-of-an-attack/

Advanced Persistent Threats

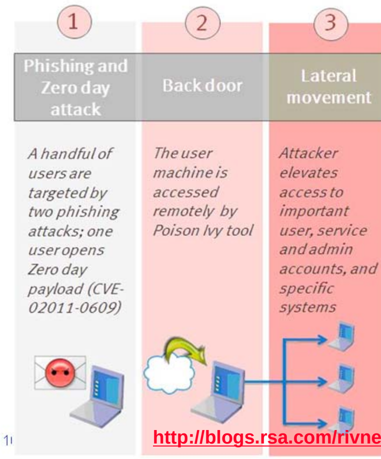
http://blogs.rsa.com/rivner/anatomy-of-an-attack/

Advanced Persistent Threats



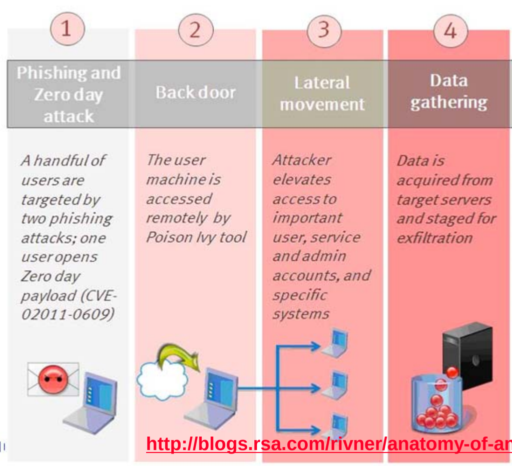
http://blogs.rsa.com/rivner/anatomy-of-an-attack/

Advanced Persistent Threats



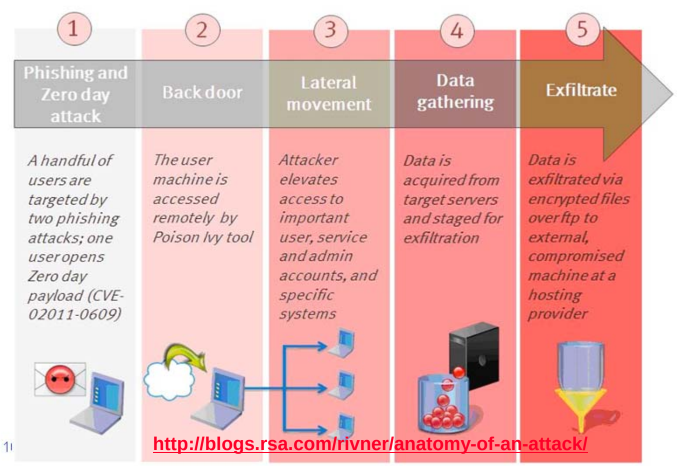
http://blogs.rsa.com/rivner/anatomy-of-an-attack/

Advanced Persistent Threats



http://blogs.rsa.com/rivner/anatomy-of-an-attack/

Advanced Persistent Threats



http://blogs.rsa.com/rivner/anatomy-of-an-attack/

Summary

- Security is one of the biggest problems today
- Host Compromise
  – Poorly written software
  – Partial solutions: better OS security architecture, type-safe languages, firewalls
- Denial-of-Service
  – No easy solution: DoS can happen at many levels
  – DDoS attacks can be very difficult to defeat

Additional Notes on Public Key Cryptography (Not required for Final Exam)

Generating Public and Private Keys

- Choose two large prime numbers p and q (>1500 256 bit long) and multiply them: $n = p*q$
- Chose encryption key e such that e and $(p-1)*(q-1)$ are relatively prime
- Compute decryption key d as
  $d = e^{-1} \bmod ((p-1)*(q-1))$
  (equivalent to $d*e = 1 \bmod ((p-1)*(q-1))$)
- Public key consist of pair (n, e)
- Private key consists of pair (d, n)

RSA Encryption and Decryption

- Encryption of message block *m*:
  – $c = m^e \bmod n$
- Decryption of ciphertext *c*:
  – $m = c^d \bmod n$

Example (1/2)

- Choose p = 7 and q = 11 → n = p*q = 77
- Compute encryption key e: (p-1)*(q-1) = 6*10 = 60 → chose e = 13 (13 and 60 are relatively prime numbers)
- Compute decryption key d such that 13*d = 1 mod 60 → d = 37 (37*13 = 481)

Example (2/2)

- n = 77; e = 13; d = 37
- Send message block m = 7
- Encryption: $c = m^e \bmod n = 7^{13} \bmod 77 = 35$
- Decryption: $m = c^d \bmod n = 35^{37} \bmod 77 = 7$

Properties

- Confidentiality
- A receiver *A* computes *n, e, d*, and sends out (*n, e*)
  - Everyone who wants to send a message to *A* uses (*n, e*) to encrypt it
- How difficult is to recover *d* ? (Someone that can do this can decrypt any message sent to *A*!)
- Recall that

  $d = e^{-1} \bmod ((p-1)*(q-1))$
- So to find *d*, you need to find primes factors *p* and *q*
  - This is provable hard