CS162
Operating Systems and Systems Programming
Lecture 26

Protection and Security II,

April 29, 2010

Ion Stoica

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

Review: How easy to guess a password?

- Ways of Compromising Passwords
  - Password Guessing:
    - » Often people use obvious information like birthday, favorite color, girlfriend's name, etc…
  - Dictionary Attack:
    - » Work way through dictionary and compare encrypted version of dictionary words with entries in /etc/passwd
  - Dumpster Diving:
    - » Find pieces of paper with passwords written on them
    - » (Also used to get social-security numbers, etc)
- Paradox:
  - Short passwords are easy to crack
  - Long ones, people write down!
- Technology means we have to use longer passwords
  - UNIX initially required lowercase, 5-letter passwords: total of 26^5=10million passwords
    - » In 1975, 10ms to check a password→1 day to crack
    - » In 2005, .01μs to check a password→0.1 seconds to crack
  - Takes less time to check for all words in the dictionary!

Review: Making password harder to crack

- How can we make passwords harder to crack?
  - Can't make it impossible, but can help
- Technique 1: Extend everyone's password with a unique number (stored in password file)
  - Called "salt". UNIX uses 12-bit "salt", making dictionary attacks 4096 times harder
  - Without salt, would be possible to pre-compute all the words in the dictionary hashed with the UNIX algorithm: would make comparing with /etc/passwd easy!
- Technique 2: Require more complex passwords
  - Make people use at least 8-character passwords with upper-case, lower-case, and numbers
    - » 70^8=6x10^{14}=6million seconds=69 days@0.01μs/check
  - Unfortunately, people still pick common patterns
    - » e.g. Capitalize first letter of common word, add one digit

Review: Making password harder to crack (con't)

- Technique 3: Delay checking of passwords
  - If attacker doesn't have access to /etc/passwd, delay every remote login attempt by 1 second
  - Makes it infeasible for rapid-fire dictionary attack
- Technique 4: Assign very long passwords
  - Long passwords or pass-phrases can have more entropy (randomness→harder to crack)
  - Give everyone a smart card (or ATM card) to carry around to remember password
    - » Requires physical theft to steal password
    - » Can require PIN from user before authenticates self
  - Better: have smartcard generate pseudorandom number
    - » Client and server share initial seed
    - » Each second/login attempt advances to next random number
- Technique 5: "Zero-Knowledge Proof"
  - Require a series of challenge-response questions
    - » Distribute secret algorithm to user
    - » Server presents a number, say "5"; user computes something from the number and returns answer to server
    - » Server never asks same "question" twice
  - Often performed by smartcard plugged into system

Goals for Today

- Distributed Authorization/Remote Storage
- Buffer overflow
- Worms and Viruses

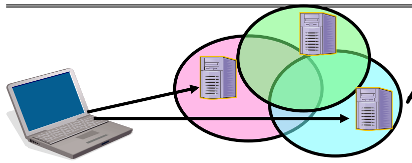
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Also, slides adapted from Kubiatowicz and Paxson.

Authorization: Who Can Do What?

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- How do we decide who is authorized to do actions in the system?
- Access Control Matrix: contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of permissions
    - » E.g. above: User D_3 can read F_2 or execute F_3
  - In practice, table would be huge and sparse!
- Two approaches to implementation
  - Access Control Lists: store permissions with each object
    - » Still might be lots of users!
    - » UNIX limits each file to: r,w,x for owner, group, world
    - » More recent systems allow definition of groups of users and permissions for each group
  - Capability List: each process tracks objects has permission to touch
    - » Popular in the past, idea out of favor today
    - » Consider page table: Each process has list of pages it has access to, not each page has list of processes …

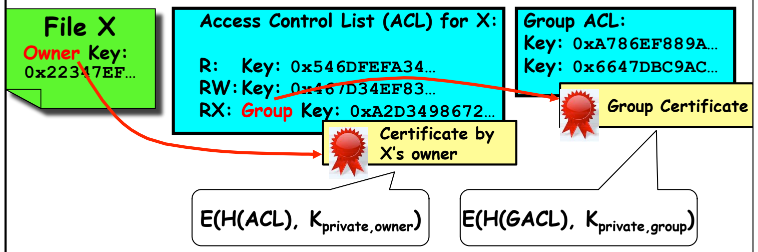
How to perform Authorization for Distributed Systems?



Different Authorization Domains

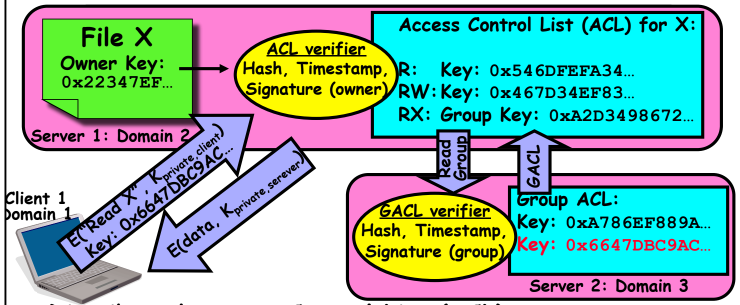
- Issues: Are all user names in world unique?
  - No! They only have small number of characters
  - Need something better, more unique to identify person
- Suppose want to connect with any server at any time?
  - Need an account on every machine! (possibly with different user name for each account)
  - OR: Need to use something more universal as identity
    - » Public Keys! (Called "Principles")
    - » People *are* their public keys

Distributed Access Control



- Distributed Access Control List (ACL)
  - Contains list of attributes (Read, Write, Execute, etc) with attached identities (Here, we show public keys)
    - » ACLs signed by owner of file, only changeable by owner
    - » Group lists signed by group key
  - ACLs can be on different servers than data
    - » Signatures allow us to validate them
    - » ACLs could even be stored separately from verifiers

Distributed Access Control



- Distributed Access Control List (ACL)
  - Contains list of attributes (Read, Write, Execute, etc) with attached identities (Here, we show public keys)
    - » ACLs signed by owner of file, only changeable by owner
    - » Group lists signed by group key
  - ACLs can be on different servers than data
    - » Signatures allow us to validate them
    - » ACLs could even be stored separately from verifiers

Analysis of Previous Scheme

- Positive Points:
  - Identities checked via signatures and public keys
    - » Client can't generate request for data unless they have private key to go with their public identity
    - » Server won't use ACLs not properly signed by owner of file
  - No problems with multiple domains, since identities designed to be cross-domain (public keys domain neutral)
- Revocation:
  - What if someone steals your private key?
    - » Need to walk through all ACLs with your key and change…!
    - » This is very expensive
  - Have unique string identifying you that people place into ACLs
    - » Then, ask Certificate Authority to give you a certificate matching unique string to your current public key
    - » Client Request: E(request + unique ID, $K_{private,client}$); give server certificate if they ask for it.
    - » Key compromise⇒must distribute "certificate revocation", since can't wait for previous certificate to expire.
  - What if you remove someone from ACL of a given file?
    - » If server caches old ACL, then person retains access!
    - » Here, cache inconsistency leads to security violations!

Analysis Continued

- Who signs the data?
  - Or: How does client know they are getting valid data?
  - Signed by server?
    - » What if server compromised? Should client trust server?
  - Signed by owner of file?
    - » Better, but now only owner can update file!
    - » Pretty inconvenient!
  - Signed by group of servers that accepted latest update?
    - » If must have signatures from all servers ⇒ Safe, but one bad server can prevent update from happening
    - » Instead: ask for a threshold number of signatures
    - » Byzantine agreement can help here
- How do you know that data is up-to-date?
  - Valid signature only means data is valid
  - Freshness attack:
    - » Malicious server returns old data instead of recent data
    - » Problem with both ACLs and data
    - » E.g.: you just got a raise, but enemy breaks into a server and prevents payroll from seeing latest version of update
  - Hard problem
    - » Needs to be fixed by invalidating old copies or having a trusted group of servers (Byzantine Agreement?)

Administrivia

- Final Exam
  - 105 Stanley Hall
  - Friday, May 14, 7:00PM-10:00PM
  - All material from the course
    - » With slightly more focus on second half, but you are still responsible for all the material
  - Closed books, two sheets of notes, both sides
- Should be working on Project 4
  - Final Project due on Friday 5/7
- I will have office hours next week at normal time
  - Tuesday & Thursday: 2-3pm

Enforcement

- Enforcer checks passwords, ACLs, etc
  - Makes sure the only authorized actions take place
  - Bugs in enforcer⇒things for malicious users to exploit
- In UNIX, superuser can do anything
  - Because of coarse-grained access control, lots of stuff has to run as superuser in order to work
  - If there is a bug in any one of these programs, you lose!
- Paradox
  - Bullet-proof enforcer
    - » Only known way is to make enforcer as small as possible
    - » Easier to make correct, but simple-minded protection model
  - Fancy protection
    - » Tries to adhere to principle of least privilege
    - » Really hard to get right
- Same argument for Java or C++: What do you make private vs public?
  - Hard to make sure that code is usable but only necessary modules are public
  - Pick something in middle? Get bugs and weak protection!

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
  - E.g., 2006 study found 15% of popular P2P files infected by one of 52 different viruses

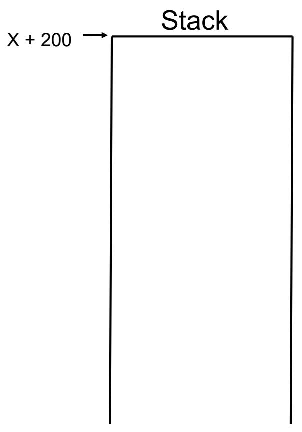
Buffer Overflow

- 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

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

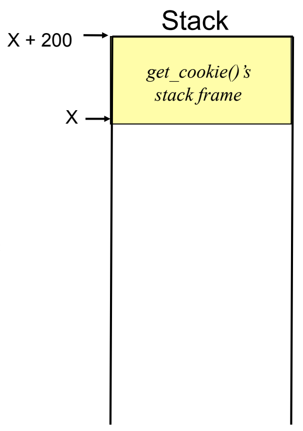
Example: Normal Execution

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



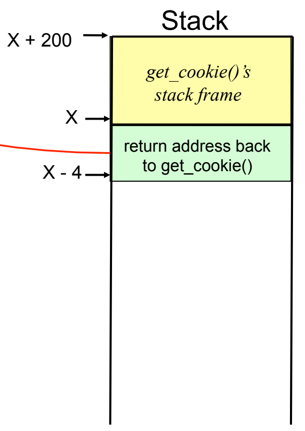
Example: Normal Execution

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



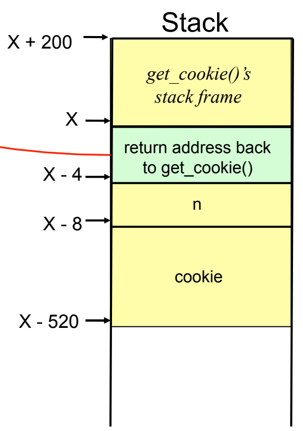
Example: Normal Execution

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


Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4

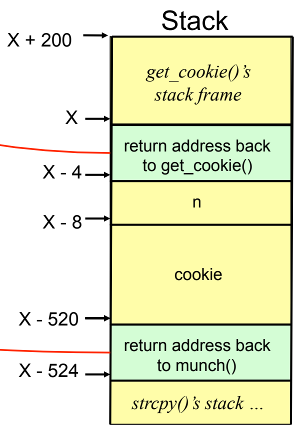
Example: Normal Execution

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


Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4
n
X - 8
cookie
X - 520

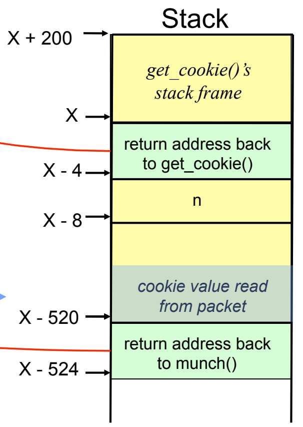
Example: Normal Execution

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


Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4
n
X - 8
cookie
X - 520
return address back to munch()
X - 524
strcpy()'s stack ...

Example: Normal Execution

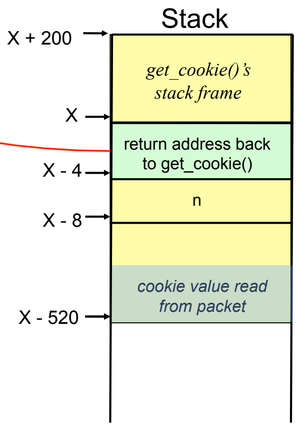
```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```



Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4
n
X - 8
cookie value read from packet
X - 520
return address back to munch()
X - 524

Example: Normal Execution

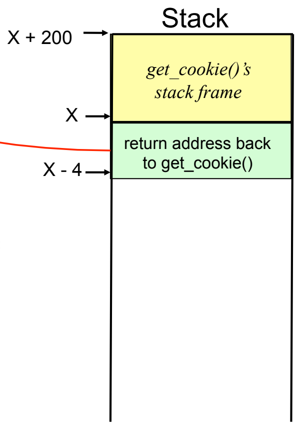
```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```



Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4
n
X - 8
cookie value read from packet
X - 520

Example: Normal Execution

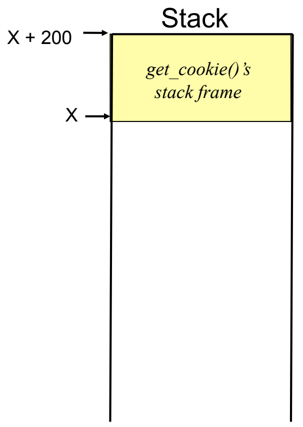
```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```



Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4

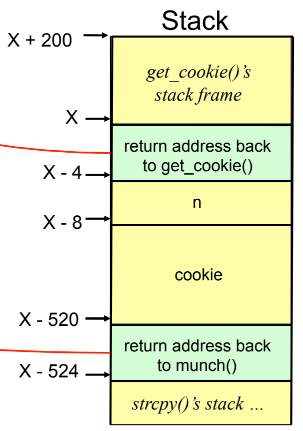
Example: Normal Execution

```
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

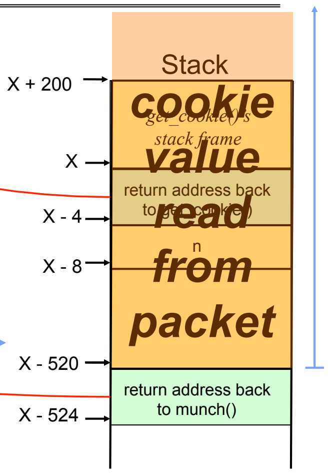
```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```



Stack
X + 200
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4
n
X - 8
cookie
X - 520
return address back to munch()
X - 524
strcpy()'s stack …

Example: Buffer Overflow

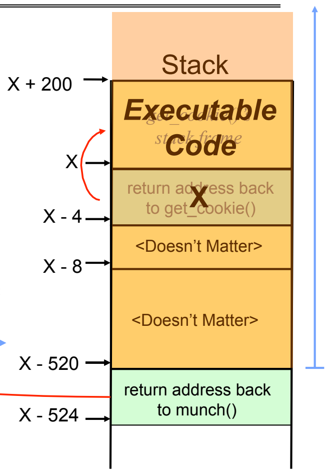
```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```



Stack
X + 200
cookie value read from packet
get_cookie()'s stack frame
X
return address back to get_cookie()
X - 4
n
X - 8
X - 520
return address back to munch()
X - 524

Example: Buffer Overflow

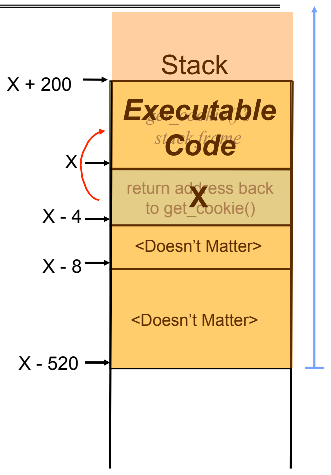
```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```



Stack
X + 200
Executable Code
get_cookie()'s stack frame
X
return address back to get_cookie()
X
X - 4
<Doesn't Matter>
X - 8
<Doesn't Matter>
X - 520
return address back to munch()
X - 524

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]);
  . . .
}
```

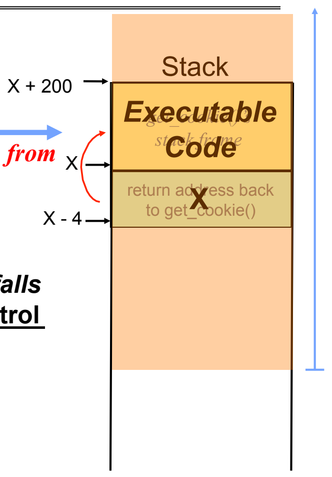


Example: Buffer Overflow

```
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


Stack
X + 200
Executable Code
get_cookie()'s stack frame
X
return address back to get_cookie()
X
X - 4

Buffer Overflows: Potential Solutions

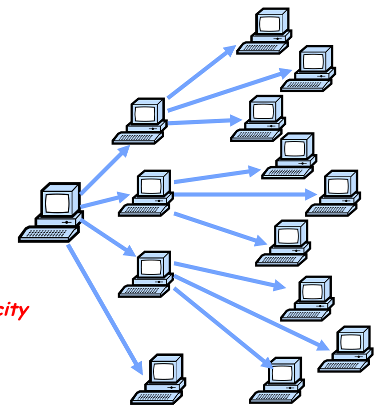
- Don't write buggy software
  - It's not like people try to write buggy software
- Type-safe Languages
  - Unrestricted memory access of *C/C++* contributes to problem
  - Use Java, Perl, Python instead
- OS architecture
  - Compartmentalize programs better, so one compromise doesn't compromise the entire system
  - E.g., DNS server doesn't need total system access
- Firewalls - restrict remote access to services
- *Intrusion detection*: recognize attack & block it

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

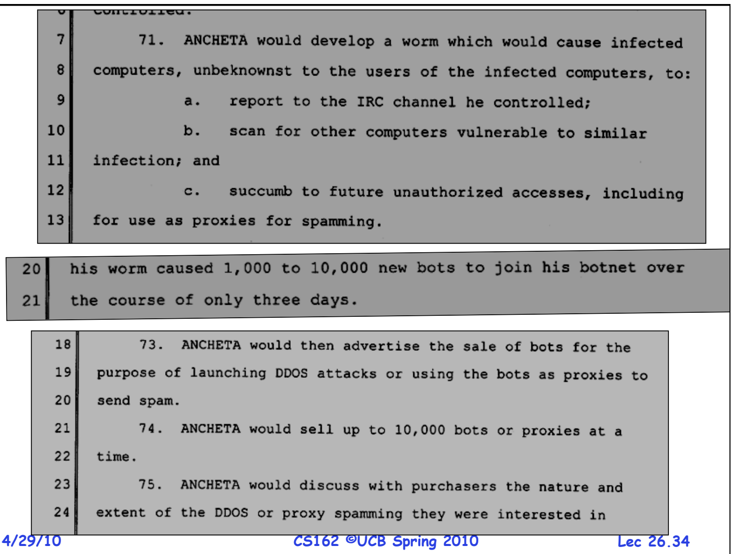
Worms: Exponentially Fast …. and Big

- 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)
  - $50B+ damage (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)


controlled.

71. ANCHETA would develop a worm which would cause infected computers, unbeknownst to the users of the infected computers, to:

a. report to the IRC channel he controlled;

b. scan for other computers vulnerable to similar infection; and

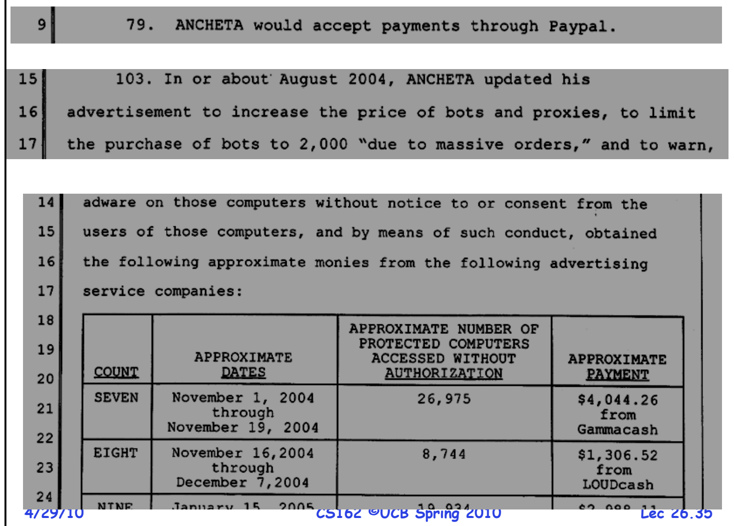
c. succumb to future unauthorized accesses, including for use as proxies for spamming.

his worm caused 1,000 to 10,000 new bots to join his botnet over the course of only three days.

73. ANCHETA would then advertise the sale of bots for the purpose of launching DDOS attacks or using the bots as proxies to send spam.

74. ANCHETA would sell up to 10,000 bots or proxies at a time.

75. ANCHETA would discuss with purchasers the nature and extent of the DDOS or proxy spamming they were interested in


79. ANCHETA would accept payments through Paypal.

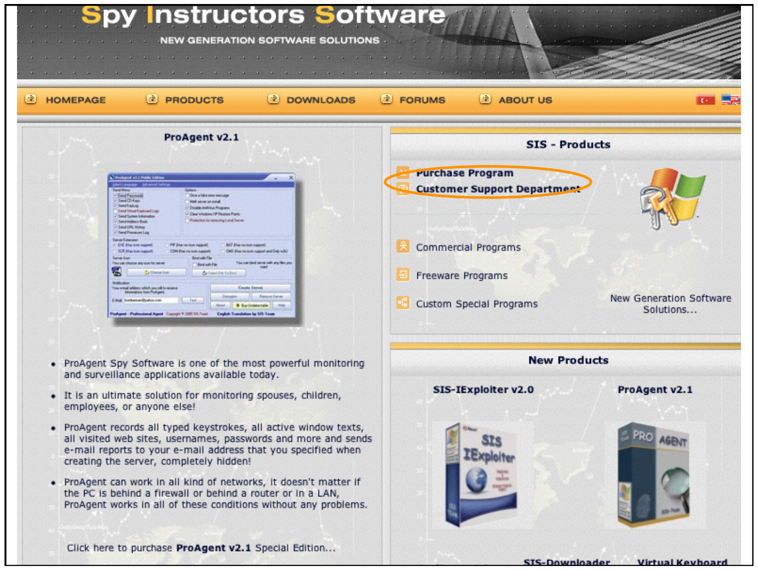
103. In or about August 2004, ANCHETA updated his advertisement to increase the price of bots and proxies, to limit the purchase of bots to 2,000 "due to massive orders," and to warn,

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:

COUNT	APPROXIMATE DATES	APPROXIMATE NUMBER OF PROTECTED COMPUTERS ACCESSED WITHOUT AUTHORIZATION	APPROXIMATE PAYMENT
SEVEN	November 1, 2004 through November 19, 2004	26,975	$4,044.26 from Gammacash
EIGHT	November 16,2004 through December 7,2004	8,744	$1,306.52 from LOUDcash
NINE	January 15, 2005	10,934	$2,988.11

Some other Attacks

- Trojan Horse Example: Fake Login
  - Construct a program that looks like normal login program
  - Gives "login:" and "password:" prompts
    - » You type information, it sends password to someone, then either logs you in or says "Permission Denied" and exits
  - In Windows, the "ctrl-alt-delete" sequence is supposed to be really hard to change, so you "know" that you are getting official login program
- Salami attack: Slicing things a little at a time
  - Steal or corrupt something a little bit at a time
  - E.g.: What happens to partial pennies from bank interest?
    - » Bank keeps them! Hacker re-programmed system so that partial pennies would go into his account.
    - » Doesn't seem like much, but if you are large bank can be millions of dollars
- Eavesdropping attack
  - Tap into network and see everything typed
  - Catch passwords, etc
  - Lesson: never use unencrypted communication!



ProAgent v2.1

- ProAgent Spy Software is one of the most powerful monitoring and surveillance applications available today.
- It is an ultimate solution for monitoring spouses, children, employees, or anyone else!
- ProAgent records all typed keystrokes, all active window texts, all visited web sites, usernames, 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...

SIS - Products

Purchase Program
Customer Support Department
Commercial Programs
Freeware Programs
Custom Special Programs

New Generation Software Solutions...

New Products

SIS-IExploiter v2.0 **ProAgent v2.1**

SIS-Downloader **Virtual Keyboard**

Conclusion

- Passwords
  - Encrypt them to help hid them
  - Force them to be longer/not amenable to dictionary attack
  - Use zero-knowledge request-response techniques
- Distributed storage example
  - Revocation: How to remove permissions from someone?
  - Integrity: How to know whether data is valid
  - Freshness: How to know whether data is recent
- Buffer-Overflow Attack: exploit bug to execute code
  - Worms spread exponentially fast
- Defenses: type-safe languages, program compartmentalization, firewalls, intrusion detection
- Many crooks out there seek ways to misuse the Internet towards their gain