CSC 369

Week 13: Security

University of Toronto, Department of Computer Science

Security

- Types of security threats
- Protection basics
- Intrusion
- Malicious software

- Some principles of computer security

Take Aways

- Security issues should not be an after thought
  - Must be considered from the beginning of a project
- Do not write your own crypto
  - Corollary: Don't trust an encryption algorithm that has not been made public
- Make security easy for your users
- Social engineering exploits are a very real problem

Security

- Computer Security
  - Techniques for computing in the presence of adversaries
    - Four requirements of security
      - Confidentiality: preventing unauthorized release of info
      - Integrity: preventing unauthorized modification of info
      - Availability: ensuring access to legitimate users
      - Authenticity: verifying the identify of a user
    - Protection is about providing all on a single machine
      - Usually considered the responsibility of the OS
- Cryptography
  - Techniques for communicating in the presence of adversaries
- Privacy
  - Policy decisions

Types of Threats

- Security requirements are intended to thwart four corresponding types of threats
- Interception, or eavesdropping - attacker gains knowledge they should not have access to
    - Loss of confidentiality
    - Reading or copying files that attacker should not have access to
    - Intercepting network packets
        - IP packets include destination field in header. Well-behaved network cards ignore packets that are not intended for them, but network cards can be placed in “promiscuous mode” where all packets are accepted

More threats…

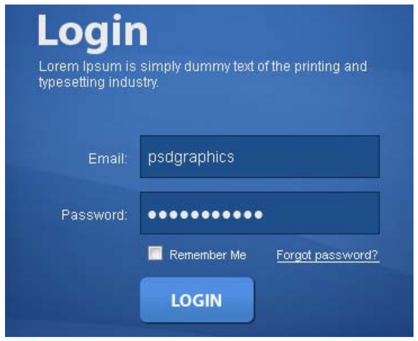
- Modification - attacker alters existing files, programs, network packets, etc.
  - Attack on integrity
  - Destruction of data is a special case
    - Attack on availability
- Theft of service – attacker installs daemon
  - Attack on availability
- Fabrication - attacker creates counterfeit objects (files, messages, etc.) which appear to come from a trusted source
  - Attack on authenticity

Vulnerabilities in the System

- Physical access
  - Unauthorized physical access makes it a lot easier to gain unauthorized digital access
- Humans
  - Who should you trust? How much?
- Operating system
  - Flaws in the system allows security protocols to be circumvented
- Networks
  - Data travels over unsecured communication lines, across multiple administrative domains
- Other software
  - BIOS, bootloader, libraries, drivers, device firmware

Authentication

- Based on one of four general principals:
- Identify
  - Something the user *knows*
  - Something the user *has*
  - Something the user *is*
  - Something the user *does*



Login

Lorem Ipsum is simply dummy text of the printing and typesetting industry.

Email: psdgraphics

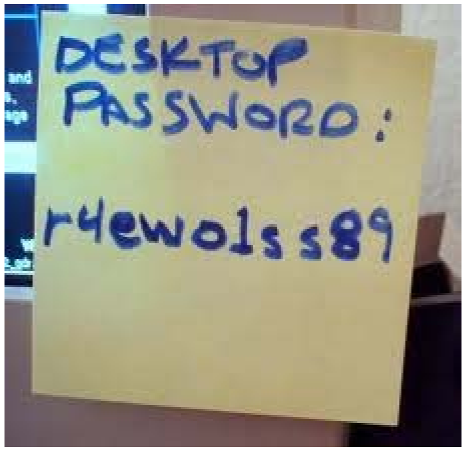
Password: ●●●●●●●●●●●

Password Authentication

- User types login name and password
- Computer system checks that the entered password matches stored password for that login name
- Depends on keeping passwords secret!
- Good password schemes make it harder for attacker to guess login information
  - Don't display success/failure after only login name is entered
  - Don't display password in clear as it is being typed
  - Don't display number of characters in the password!
  - Force users to choose strong passwords
  - Force users to change passwords frequently
  - Disallow reuse of last N passwords

Security vs. Usability

- Users need to remember their passwords
- Strict password rules make this harder



- Lots of website logins (should) mean lots of passwords to remember
    - Users commonly choose the same password for multiple sites

Storing Passwords

- Where/how should the system store password file?
- Option 1: On disk, protected by file access restrictions
  - Too many people and programs have access
- Option 2: Store function of password on disk
  - Password file contains <login_id, E(pwd, fixed data)>
  - Can be publicly readable
  - Login program compares E(entered_pwd, fixed data) with stored value for login_id
  - But attacker can encrypt likely passwords offline and compare with publicly-readable values in password file
- Option 3: Add random “salt” to password, E(pwd+salt,…)
- And restrict access to sensitive part of password file

One-Time Passwords (OTP)

- Password is only valid for one login
- No risk of password compromise!
- How do computer system and user both know the next password to use?
  - Time-synchronized security token given to user
    - Current time is used to generate next password
  - Mathematical function, one-way hash chain
    - User picks secret password, s, number of OTPs, e.g. 4
    - P_1 = f(f(f(f(s)))), P2 = f(f(f(s))), P3 = f(f(s)), P4 = f(s)
    - P_{i-1} = f(Pi)
    - Server stores P_0 and 1 in password database, initially
    - Asks client for P_1 on login, computes P_0 to verify, stores P_1, 2

Challenge-Response Authentication

- System issues a “challenge”, user authenticates by providing correct response
- Standard password schemes are one form of this
  - But the challenge, and hence the response, are always the same
- Other examples?
  - Security questions to recover lost passwords
    - “Who was your best friend in high school?”
  - CAPTCHA to verify human user

Authentication with Physical Object

- “Something the user has”
- Ordinary door keys
- Swipe card
- Chip-and-pin credit cards
- Smart cards

Authentication Using Biometrics

- "Something a user is"
- Some physical characteristics are unique and hard to forge
  - Fingerprint
  - Voiceprint
  - Iris pattern
  - Finger length (of all fingers)
- Requires *enrollment* prior to use
  - Measure and record biometric feature
- Must be acceptable to human users
- E.g. India is issuing nation-wide Unique ID (UID)
  - Record both iris scans, all 10 fingerprints, facial photo
- "Something a user does" is a *behavioral biometric*

Protection / Access Control

So … a user has been authenticated by the system

Now what?

- Often want to restrict what a user can do
  - What files can be read / modified / removed?
  - What system settings can be changed?
  - What permissions can be changed?

- Unix philosophy: “Everything is a file”
  - Thus, protection is provided through the file system

Protection / Access Control

- File systems implement some kind of protection system
  - Who can access a file
  - How they can access it
- More generally…
  - Objects are “what”, subjects are “who”, actions are “how”
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  - You can read and/or write your files, but others cannot
  - You can read “/etc/motd”, but you cannot write it
  - Only a superuser can execute /usr/sbin/adduser

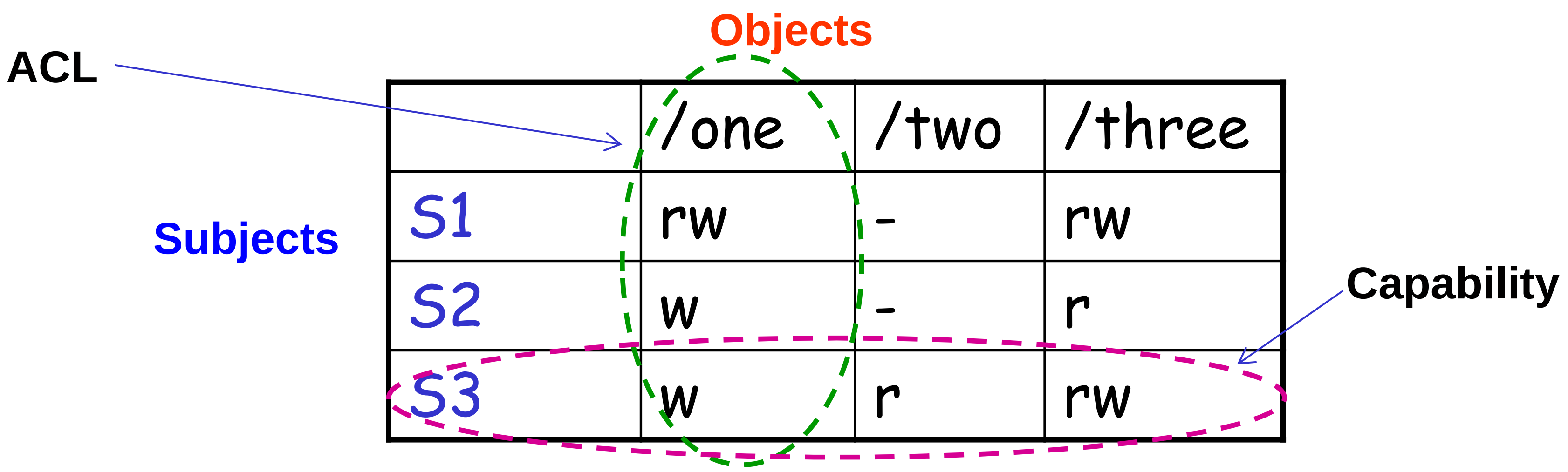
Types of Access

- None
- Knowledge
- Execution
- Reading
- Appending
- Updating
- Changing Protection
- Deletion

- Unix provides only Read/Write/Execute permissions
  - How are other permissions realized?

Representing Protection

- Lampson Access Matrix
  - Captures state of a system at some instant in time
  - Entry represents access that a subject has to an object
  - Can map to idea of *access control lists* and *capabilities*



	/one	/two	/three
S1	rw	-	rw
S2	w	-	r
S3	w	r	rw

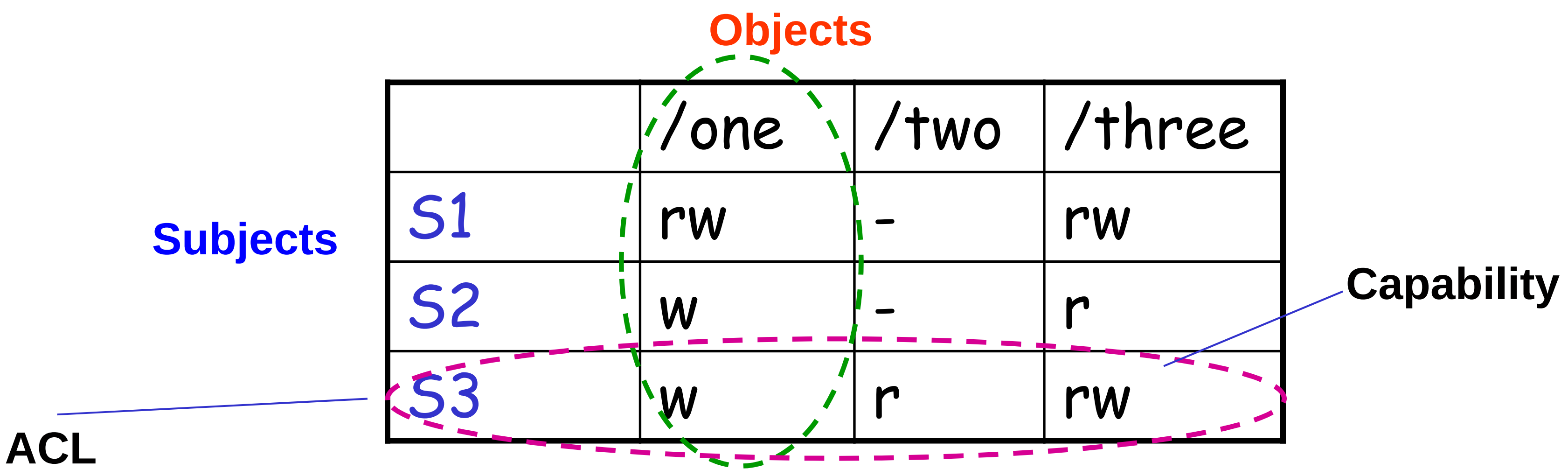
Representing Protection

Access Control Lists (ACL)

- For each object, maintain a list of subjects and their permitted actions

Capabilities

- For each subject, maintain a list of objects and their permitted actions



	/one	/two	/three
S1	rw	-	rw
S2	w	-	r
S3	w	r	rw

ACLs in Action

Assume

- 2 global mappings
  - Principal: process → user identity
  - Fs_lookup: filename → object identity
- Per-object mapping
  - ACL: (object identity, user identity) → operations

Given a process *p* that wants to perform operation *op* on object *o*, check predicate

op in ACL(*o*, Principal(*p*))

Capabilities in Action

Assume

- 1 local per-process mapping
  - Cap: (process identity, *index*) → capability
- 2 “accessor” functions
  - Obj: capability → object identity
  - Ops: capability → operations

Given a process *p* that wants to perform operation *op* on object *o*, *p* must possess capability naming *o* at some index, *i*

o == Obj(Cap(*p*,*i*)) for some *I*

To perform operation, *p* gives index for its capability, check

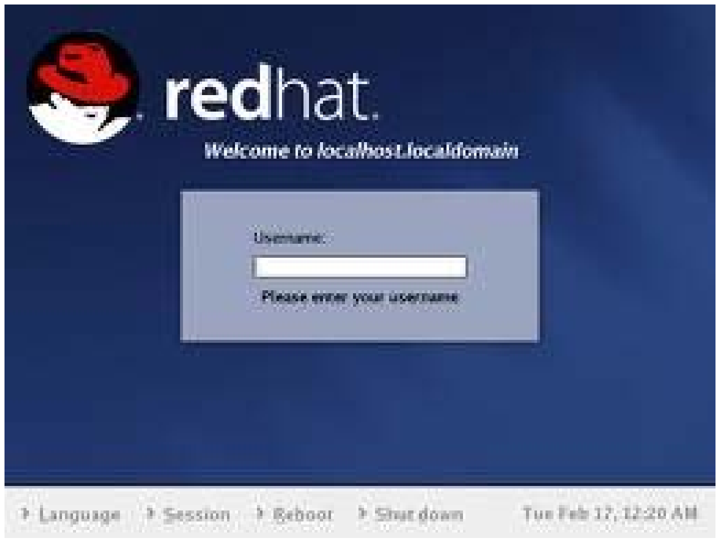
op in Ops(Cap(*p*, *i*))

ACLs vs. Capabilities

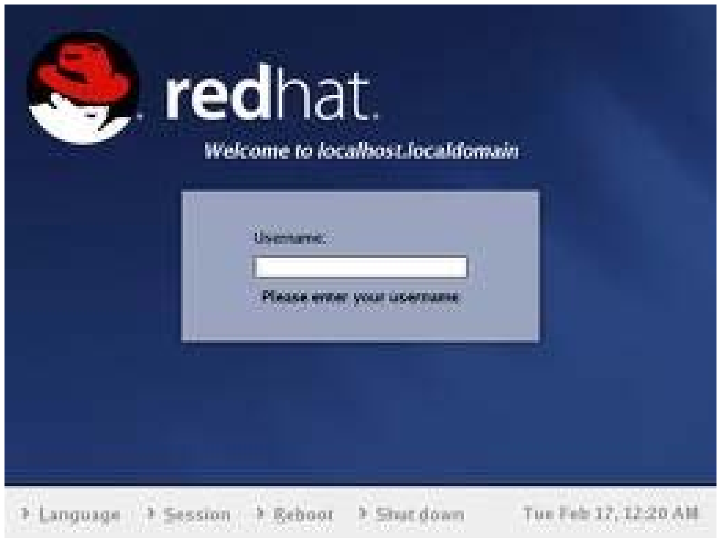
- In practice, ACLs are easier to manage
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
  - Easy to make mappings persistent (store with inode in FS)
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
- ACLs have a problem when objects are heavily shared
  - The ACLs become very large
  - Use groups (e.g., Unix)
- Use combination in many real systems
  - e.g. ACL for checking open(), capability (the file descriptor) for read / write

Insider Attacks

- *Login Spoofing*



Regular login screen



Full-screen image displayed by malicious user

Bugs

- What is a “flaw” in the operating system?
    - Code cannot be inherently flawed
    - A bug is an unintended behavior
- May be caused by
    - Bad compiler
    - Bad libraries
    - Bad user code
    - Bad input
    - Bad design
    - Hardware error
- This definition covers all possible security holes
    - A bug does not necessarily lead to a crash or a failure

Exploits

- Taking advantage of a bug to achieve something that should not normally be allowed
  - Contrary to typical policy
  - Not limited to security
- Local exploit
  - Something requiring access to the system first
    - Physical or network login
- Remote exploit
  - Leverages bug in a network service

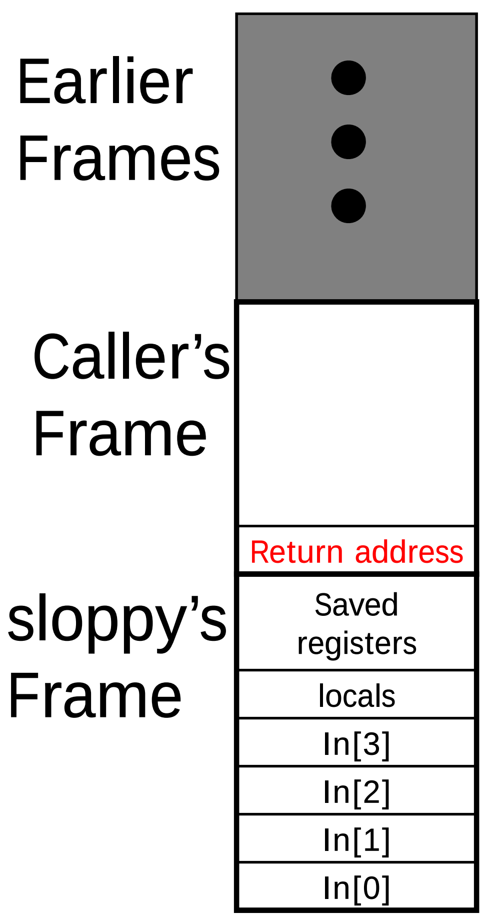
Malicious Software

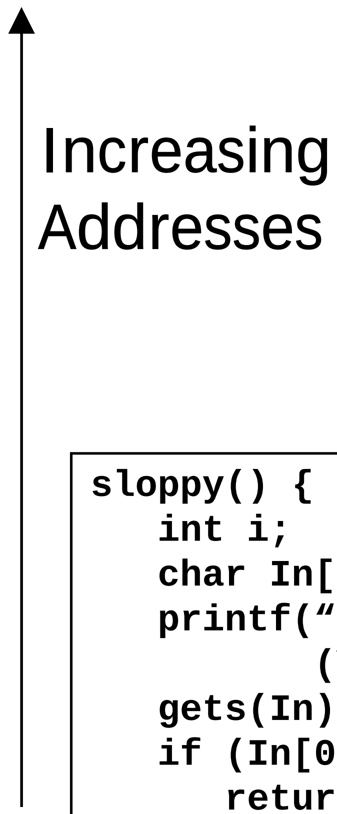
- Trap doors - program contains secret entry point that allows attacker to bypass security
- Logic bombs - destructive code in legitimate program triggered by some event
- Trojan horses - apparently useful program that tricks users into running it.
  - May contain a logic bomb, or be a vehicle for spreading a virus
- Viruses
  - A program that can “infect” other programs by copying itself into them
- Worms
  - Program that spreads via network connections

Stack & Buffer Overflow Attacks

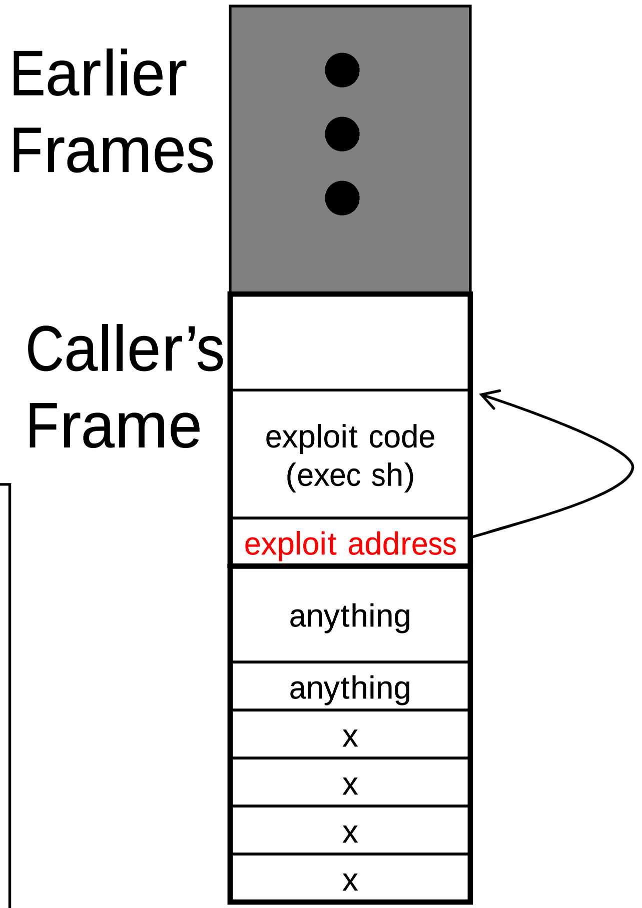
- Most common means of gaining unauthorized access to a system
- How it works:
  - Overflow some stack-allocated input buffer past the space allocated for the input
  - Overwrite the return address with the address of exploit code
  - Overwrite next space in stack with the exploit code itself
- ??? Let's see what this looks like…

Stack Attack





```
sloppy() {
   int i;
   char In[4];
   printf("Continue
          (Y/N)?\n");
   gets(In);
   if (In[0] == 'N')
      return;
   /* more code... */
}
```



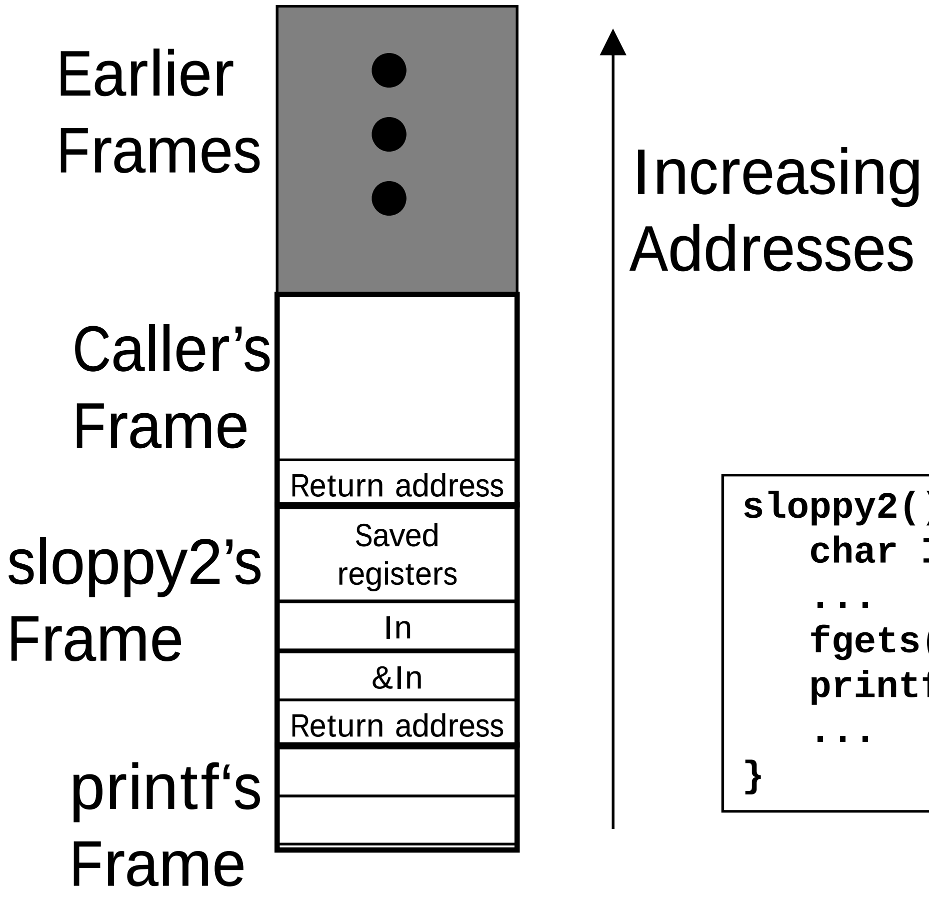
Exploiting Code Bugs

- We saw an example of *code injection* with a *buffer overflow* earlier.
  - Defense: Make stack non-executable
- Attacker need not inject new code however
  - Return-to-libc attacks
  - Overflow stack-allocated buffer to overwrite return address with address of C library function
  - Defense: Address space layout randomization

Format String Bugs

- C library formatted I/O strings can allow almost arbitrary inspection and modification of program if misused
  - First came to light ~2000
  - E.g. “printf(inputstr)” where inputstr is supplied by user
    - inputstr can contain format characters (like %x)
    - printf assumes extra arguments have been pushed on stack, dumps stack contents to output
    - %n format allows write to a memory location

Format String Attack



```
sloppy2() {
   char In[12];
   ...
   fgets(In, 12, stdin);
   printf(In);
   ...
}
```

"%x %x %x %x"

Prints stack contents

Intrusion Detection

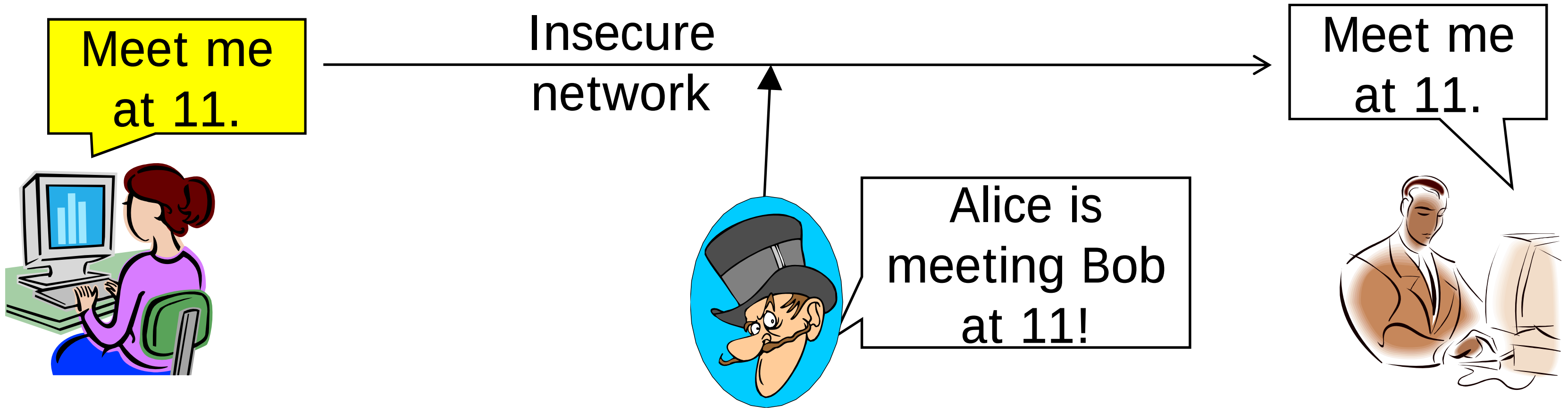
- Generally want to know when a system has been compromised
- Two main approaches
  - Signature-based detection – examine system activity or network traffic for patterns that match known attacks
  - Anomaly detection – identify patterns of "normal" behavior over time and detect deviations from those patterns
- Systems need records of user activity to help detect and recover from intrusions
  - These records have to be protected from tampering by the intruder

Network Communications

- Protecting “assets” on a single computer system with multiple users is hard, but dealing with networks is even harder
  - Multiple administrative domains
  - Users of network may not have common interests
- Two main categories of network attacks
  - Passive - eavesdropping or monitoring communications
  - Active - modification or tampering with the communication stream

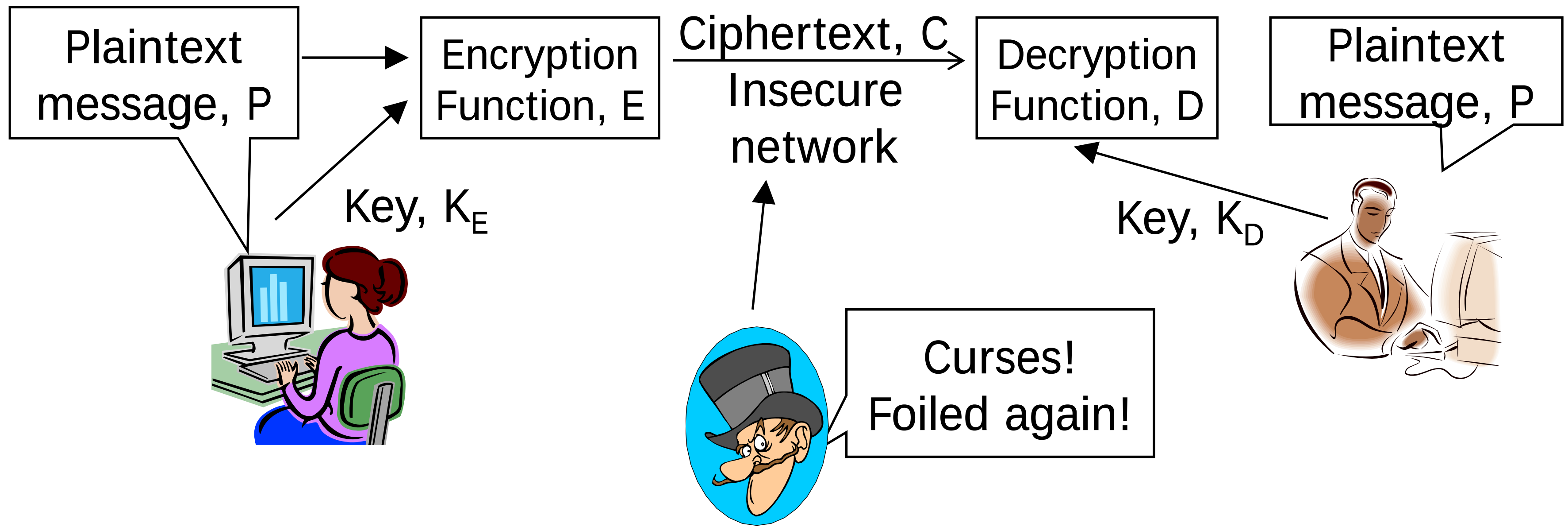
Passive Network Attacks

- *Eavesdropping* - read network packets that were not intended for attacker
  - Does not interfere with delivery of message to intended recipient



  - Defense: obfuscate message contents so attacker gains no information from intercepting the message (i.e., encrypt the message)

Encryption basics



- $C = E(P, K_E)$
- $P = D(C, K_D) = D(E(P, K_E), K_D)$

Encryption Keys

- Keys used for encryption and decryption can be the same (symmetric, or secret-key) or different (public-key)
- Secret-key cryptography
  - ✓ Fast encryption/decryption algorithms are known
  - ✗ Distributing the shared secret key is a problem
- Public-key cryptography
  - Pair of keys, one to encrypt, one to decrypt
  - ✓ Encryption key can be published, decryption key is kept secret; knowledge of one key does not reveal other
  - ✗ Encryption/decryption algorithms ~1000X slower

Diffie Hellman Merkle Key Exchange

Function: g modulo p (where p is prime)

- Alice and Bob publicly agree to use p=23, g=5
- Alice chooses secret 6
  - Sends Bob 5^{6} mod 23 = 8
- Bob chooses secret 15
  - Sends Alice 5^{15} mod 23 = 19
- Alice computes s = 19^{6} mod 23 = 2
- Bob computes s = 8^{15} mod 23 = 2
- The shared secret key is 2

Eve can know p, g, 8, 19, and it is *hard* to compute **2**

SSH

- Establish TCP connection
  - Agree on a session key
    - Server sends public RSA key (host identity)
    - Symmetric key exchange using Diffie-Hellman
    - Client gets confirmation message from server encrypted with symmetric session key
  - Authenticate client
    - password
    - Public key – client sends public key
      - Server sends challenge encrypted with public key
      - Client decrypts with private key
      - Client sends challenge back encrypted with session key

More Passive Attacks

- *Traffic Analysis* - infer information based on sender/recipient of messages, size of messages, frequency of communication, etc.
  - Encryption doesn't help here
  - Defense: obfuscate communication pattern (i.e., make all messages the same length, all communications have same number of messages, etc.)

Active Network Attacks

- *Replay* - capture a legitimate message and re-send it later to produce unauthorized effect
    - e.g. unauthorized file accesses
    - Defence: “nonce” - messages include an item so they can’t be reused
- *Modification of messages* - alter contents of message to change effect
    - Include *message digest* to detect tampering
- *Masquerade* - attacker pretends to be another entity or user
    - Often combined with another attack like replay

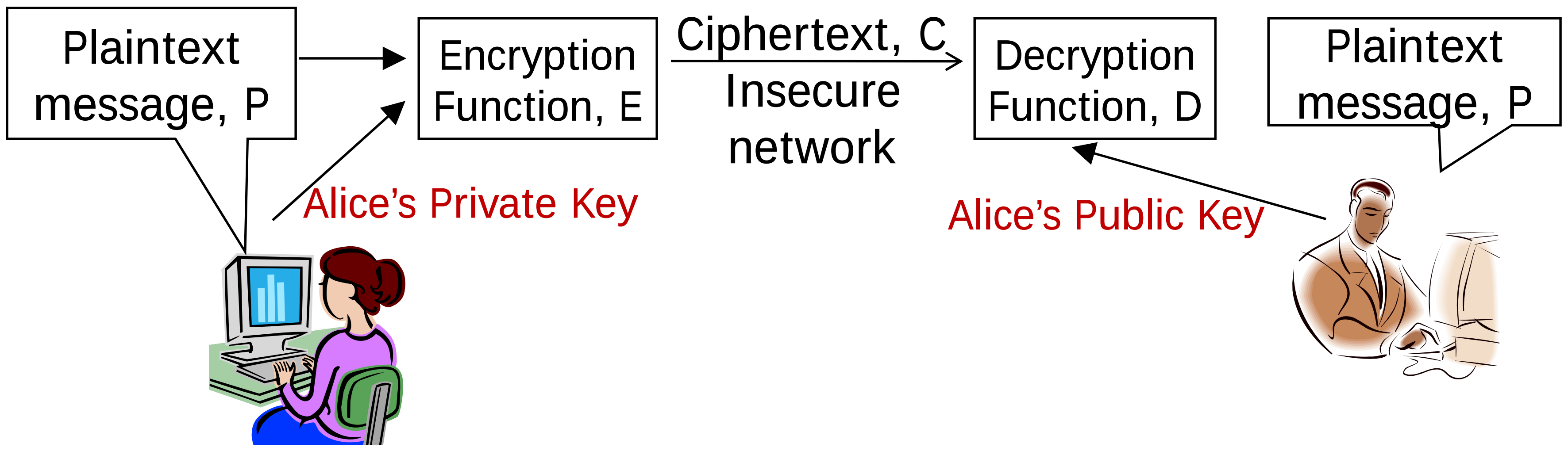
Message Digests

- Given some input data of arbitrary length, compute a fixed-length output
  - Relies on one-way function
    - Given input x, computing digest y = f(x) is “easy”
    - Given y, finding x is computationally infeasible
  - So attacker cannot construct a new input that will have the same digest as the original
- To detect tampering, compare digest computed on received message against digest of sent message
  - Need to know original digest (same problem?)
- Sender can encrypt or sign digest only
  - Or distribute digest separately from message

Digital Signatures

- Provide a way to verify the origin and integrity of a message (i.e. authenticity)
- Sender includes something with the message (or modifies the message in some way) that could only be generated by the sender
- Public-key cryptography helps here
  - Confidentiality – encrypt message with recipients public key; only intended recipient can decrypt using their private key
  - Signatures – encrypt message with sender's private key; anyone can decrypt using sender's public key, thereby verifying that sender created the message
  - Typically, only a message digest is signed to reduce overhead

Example of Signing



- Alice "signs" her message by appending a copy (or a digest) encrypted with her private key
- Bob can decrypt with Alice's public key and compare result to original message
- Bob can be sure message came from Alice

One more active attack

- Denial of service - system asset becomes unavailable to legitimate users
  - Attack on availability
  - Common network attacks - overload server with bogus requests so there are no resources for legitimate ones (e.g. TCP SYN flooding)
  - Extra evil - distributed DOS - compromise a set of computers and use them as “zombies” in a coordinated attack on the victim
  - How do you distinguish an attack from heavy load?
- Active attacks are harder to defend against
  - Need measures to detect and recover from them

Case Study: TSL

- Recall crypto basics
  - Messages sent over insecure networks are *encoded* using a secret *key*
  - Only parties with the correct key can decode the message
  - Two main types of keys
    - Symmetric, or secret key, cryptography
      - Same key is used to encode and decode
      - Encode/decode operations are computationally fast
    - Asymmetric, public key, cryptography
      - 2 keys are needed, one is kept secret, the other is public
      - Messages encoded with one key can be decoded with the other
      - Enables both privacy and authentication
      - Encode/decode operations are computationally expensive

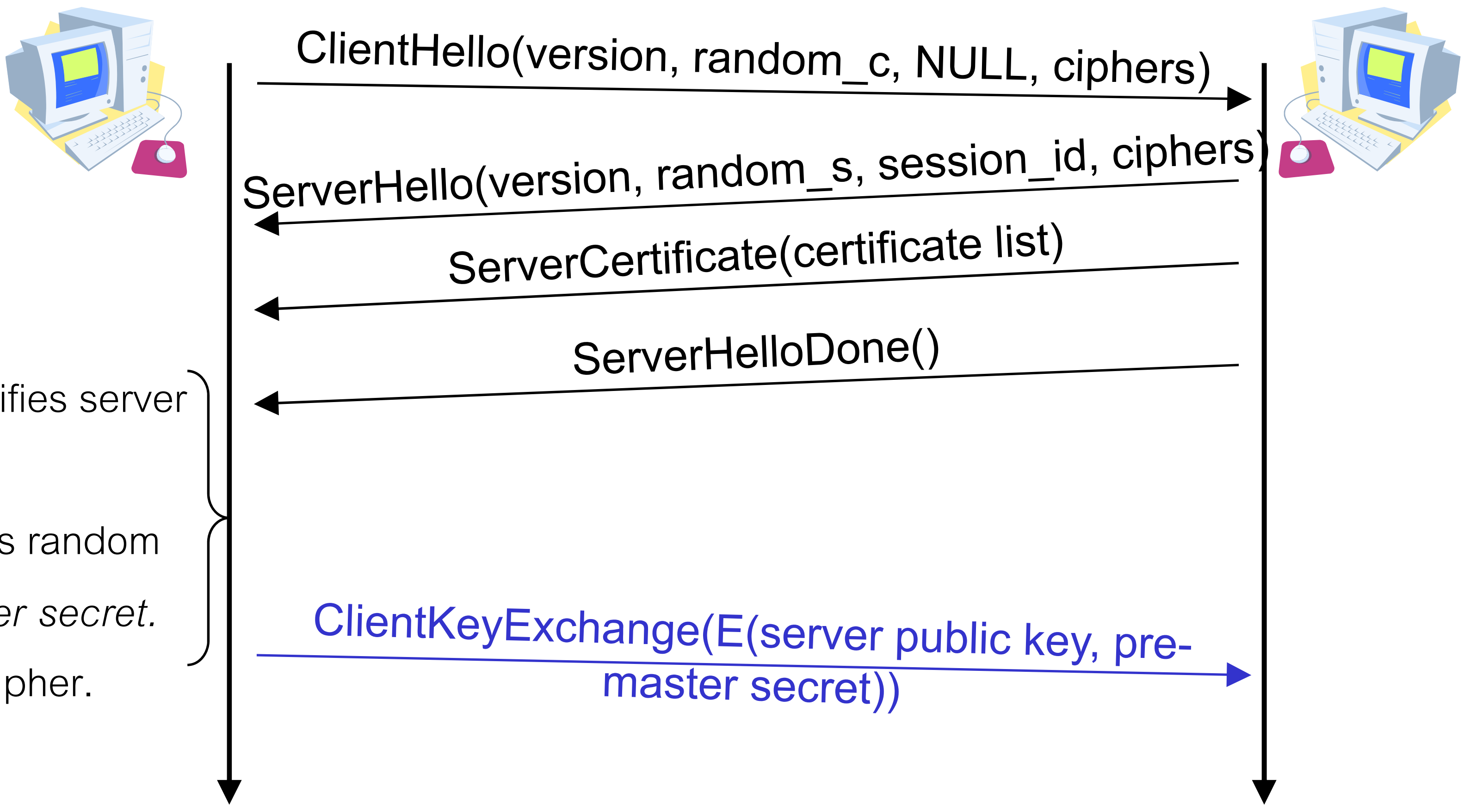
Transport Layer Security (TLS)

- TLS is the successor to SSL (Secure Sockets Layer)
  - Used for secure communications on insecure networks
  - Typically seen in web services (https://)
  - Can be used for any service above TCP/IP layer
- Provides mutual authentication of client and server
  - Client authentication is less common
- Also provides confidentiality and message integrity
- TLS uses both public key and secret key cryptography
  - Public keys are used in *handshake* to establish secret *session keys*
  - Secret keys are used in remainder of communications (faster)

TLS Communication

- Communication begins with a *handshake protocol* between client and server
  - To establish server and client identity
  - To set up *session keys* used to encrypt remainder of the transmissions
- Requires the existence of a trusted *Certification Authority (CA)*
  - CA issues certificate to verify server's identity (i.e., its name, IP address, and public key), signed with CA's private key
  - Client can verify certificate using CA's public key
    - Client issues challenge using public key, only holder of matching private key can answer correctly
  - Web applications ship with a (long) list of CA's pre-installed

TLS Full Handshake Protocol (1)



ClientHello(version, random_c, NULL, ciphers)
ServerHello(version, random_s, session_id, ciphers)
ServerCertificate(certificate list)
ServerHelloDone()
ClientKeyExchange(E(server public key, pre-master secret))

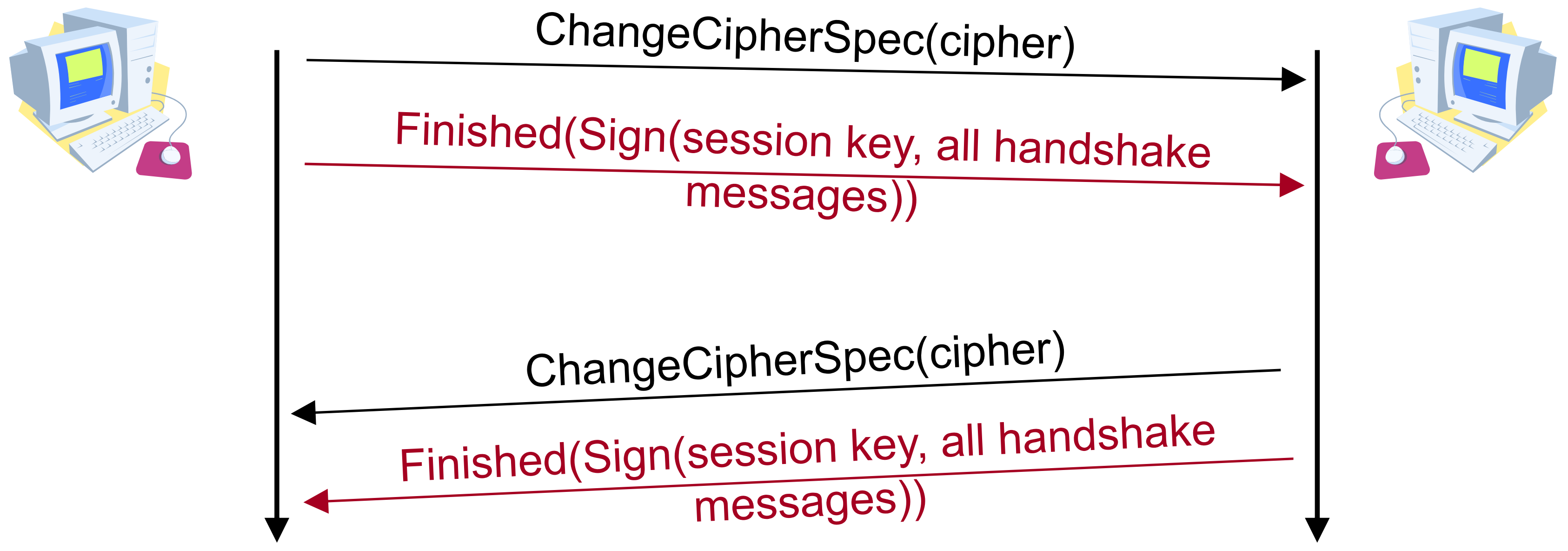
Client verifies server identity.

Generates random *pre-master secret.*

Selects cipher.

Now client & server each generate shared secret session keys using selected cipher and (random_c, random_s, pre-master secret)

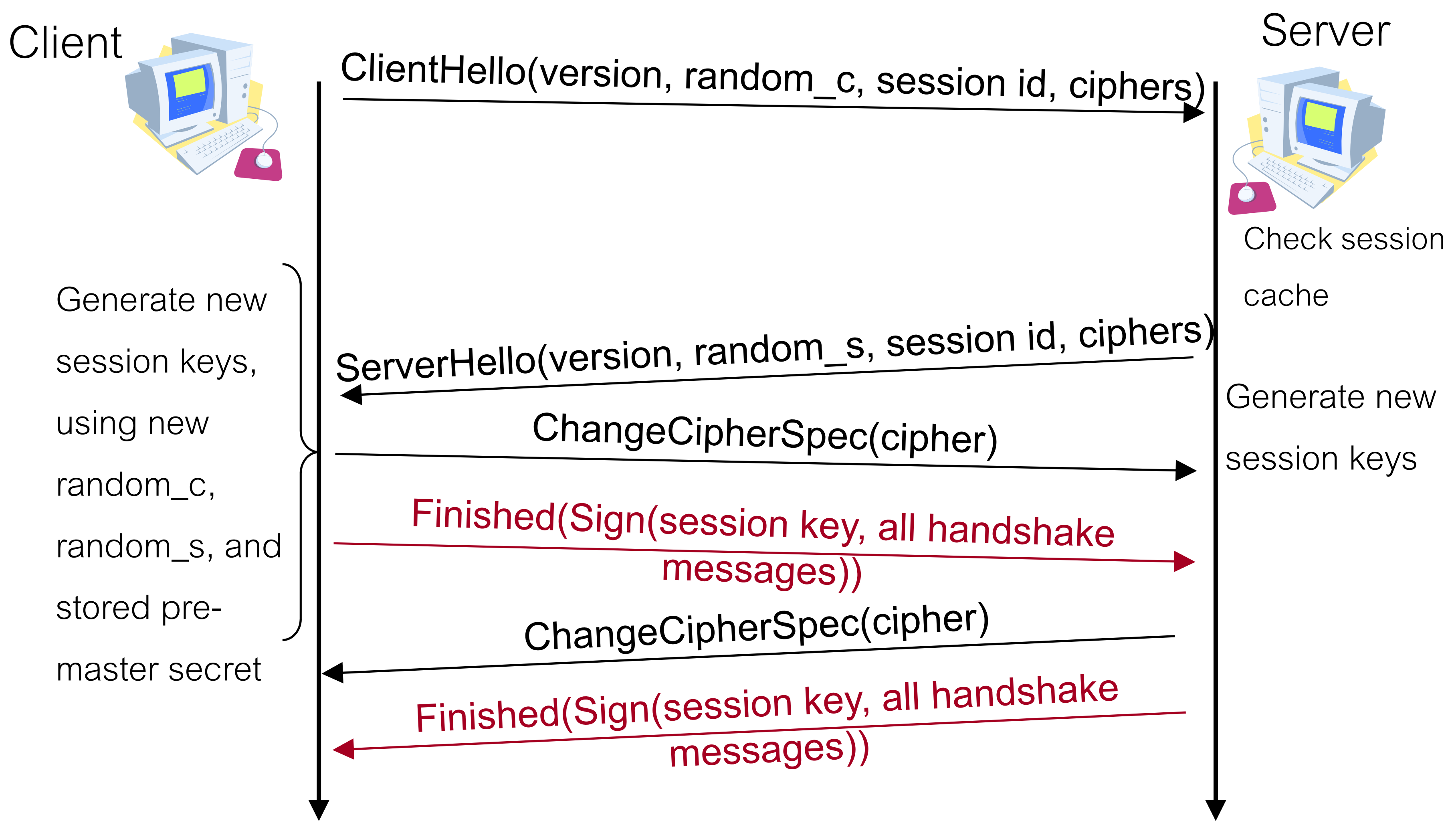
SSL Handshake Protocol (2)


ChangeCipherSpec(cipher)
Finished(Sign(session key, all handshake messages))
ChangeCipherSpec(cipher)
Finished(Sign(session key, all handshake messages))

All future communication is encrypted using session key

("Finished" messages are encrypted and signed)

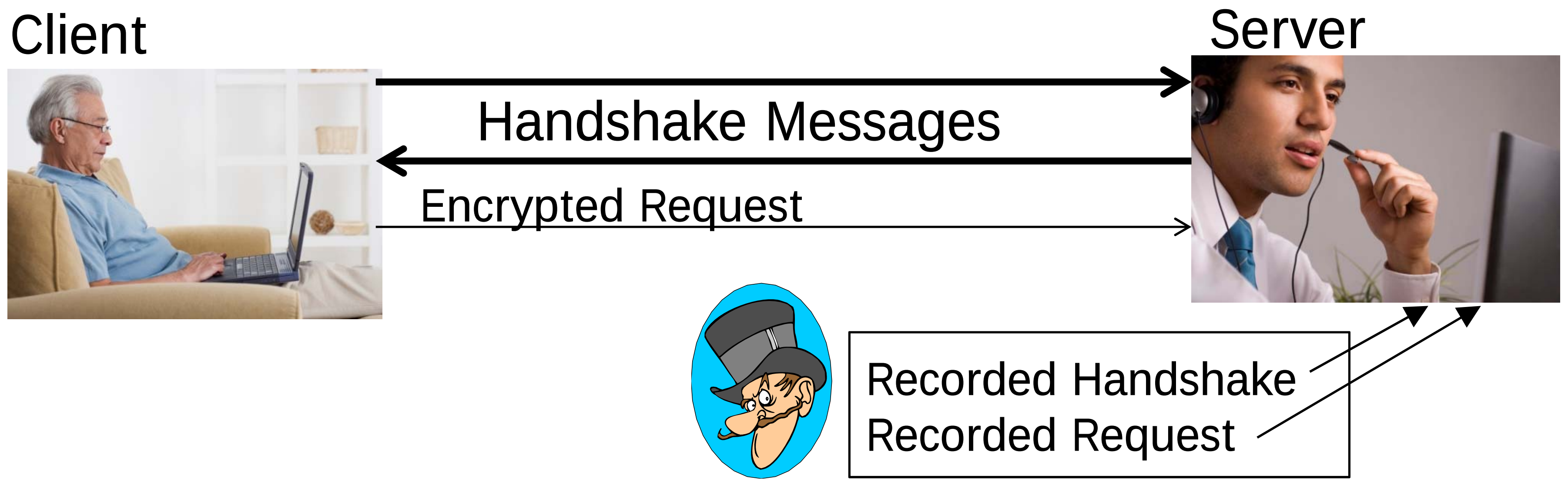
TLS Resume Handshake Protocol


Client
Server
ClientHello(version, random_c, session id, ciphers)
Check session cache
ServerHello(version, random_s, session id, ciphers)
Generate new session keys
Generate new session keys, using new random_c, random_s, and stored pre-master secret
ChangeCipherSpec(cipher)
Finished(Sign(session key, all handshake messages))
ChangeCipherSpec(cipher)
Finished(Sign(session key, all handshake messages))

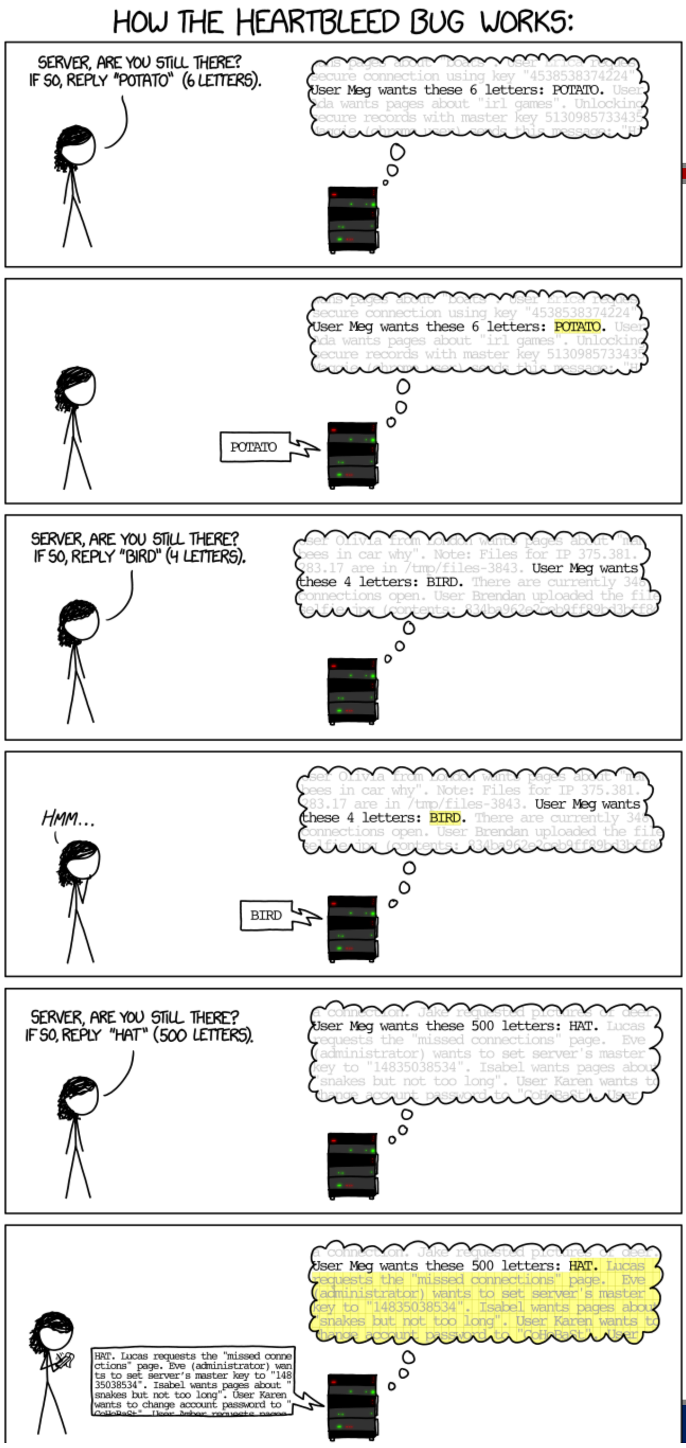
Notes on TLS Handshake Protocol

- Server certificate signed by certification authority prevents masquerade and man-in-the-middle attacks
  - No masquerade: imposter can present server's real certificate, but does not have server's private key, so can't really pretend to be server. Imposter can't present own certificate pretending to be server unless CA is compromised
  - Man-in-the-middle is a double masquerade (attacker plays role of client to server and plays role of server to client).
- Using random numbers proposed by both client and server in secret key generation prevents *replay* attacks
  - No server random: attacker can replay client handshake messages to server; server believes it is talking to original client
  - No client random: attacker can replay server's half of session resume handshake
- Using just client/server random values would allow any eavesdropper to also calculate the session keys, since they are sent in the clear. The pre-master secret means only client and server can calculate keys

Example Replay Attack



- Assume there is no server random in the protocol
  - Attacker can record session and replay handshake to establish connection with server, then replay request
  - Suppose request is payment to a merchant's paypal account
- Server random ensures each session is fresh


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Meg wants these 4 letters: BIRD.
HMM...
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT.

Heartbleed Bug

- A bug in the implementation of the Heartbeat protocol for TLS
- Heartbeat protocol: send a packet to server with an arbitrary payload. Server response tells client session is still alive.
- Bug: client sends a 2-byte payload_size. Server sends payload_size bytes back to client, even when payload_size is much larger than the payload sent by client.
- Client receives up to 64KB of memory from the server process that might include passwords, and other secure data.

https://xkcd.com/1354/

Security Design Principles

- Security is much, much more than just crypto
  - Cryptography algorithms depend on computational “hardness” of certain functions, not on keeping the algorithm itself secret
  - If there is a fundamental flaw in the design of the system, then all of the crypto in the world won’t help you
  - It is usually easier to find a bug in an implementation than to circumvent a crypto system
- Unfortunately, systems design is still as much an art as it is a science
  - But, decades of building systems the wrong way have helped us gain some learned wisdom
  - The remaining slides cover some of this, but are not on exam

Principle of Least Privilege

- Figure out exactly which capabilities a program needs to run, and grant it only those
  - Not always easy, but one algorithm: start with granting none, run and see where it breaks, add new privileges, repeat
- Unix
  - Good example: Should not normally run as root to guard against accidents
  - Bad example: Some programs run as root just to get a small privilege, such as using a port < 1024 (privileged port)
    - E.g., ftpd, httpd
    - Exploit these programs, and you get root access to system

Principle of Least-Common Mechanism

- Basic lesson: Be careful of shared code
  - Assumptions made may no longer be valid with shared code
- Eudora/Outlook and Internet Explorer
  - Windows exports an API to IE’s HTML rendering code
    - Eudora and other programs use this to display HTML in email
    - By default, JavaScript and Java parsing are enabled
  - HTML rendering code knows Java(Script) is unsafe
    - Disables it when JavaScript is downloaded from Internet
      - Internet is untrusted
    - But enables it when JavaScript is loaded off of disk
      - Your own file system is trusted
  - But…email is loaded off of disk!
    - Fertile ground for email viruses…

Principle of Complete Mediation

- Check every access to every object
  - In rare cases, can get away with less (caching)
    - But only if sure nothing relevant in environment has changed (which is a lot)
- Ex: NFS and file handles
  - NFS protocol
    - Client contacts remote “mountd” to get a file handle to a remotely exported NFS file system
      - Remote mountd checks access control at mount time
    - File handle is a capability: client presents it to read/write file
      - Access control is not checked after mount time
    - Can use network sniffer to get file handle and access file system

Consequence: TOCTTOU bugs

- NFS example is one instance of a “time-of-check-to-time-of-use” bug / security flaw
- This is also a form of *race condition*
- Consider the following code fragment from a program that runs with setuid root:

“access” checks filename for access permission using *real* user and group id, not effective id.

“open” uses effective user id (root in this case) to open the file.

```
if (access(filename, W_OK) == 0) {
      if ((fd = open(filename,O_WRONLY)) == NULL) {
            perror(filename);
            return 0;
      }
      /* Now write to the
}
```

```
Attacker:
% touch /tmp/foo
% run_priv /tmp/foo
% rm /tmp/foo
% ln /etc/passwd /tmp/foo
```

Fail-Safe Defaults

- Start by denying all access, then allow only that which has been explicitly permitted
  - Oversights will then show up as “false negatives”
    - Somebody is denied access who should have it
  - Opposites lead to “false positives”
    - Somebody is given access that shouldn’t get it
    - Not much incentive to report this kind of failure…
- Examples
  - SunOS shipped with “+” in /etc/hosts.equiv
    - Essentially lets anyone login as any local user to host
  - Irix shipped with “xhost +”
    - Any remote client can connect to local X server

No Security Through Obscurity

- Security through obscurity
  - Attempting to gain security by hiding implementation details
  - Claim: A secure system should be secure even if all implementation details are published
    - In fact, systems become more secure as people scour over implementation details and find flaws
    - Rely on mathematics and sound design to provide security
- Ex: GSM cell phones
  - GSM committee designed their own crypto algorithm, but hid it from the world
    - Social engineering + reverse engineering revealed the algorithm
    - Turned out to be relatively weak, easy to subvert

Trusted Computing Base (TCB)

- Think carefully about what you trust with your data
  - If you type your password on a keyboard, you're trusting
    - The keyboard manufacturer
    - Your computer manufacturer
    - Your OS
    - The password library
    - The application that is checking the password
  - TCB = set of components (hardware, software, people) that you trust your secrets with
- Public Web kiosks should not be in your TCB
  - Should your OS? (Think about IE and ActiveX)