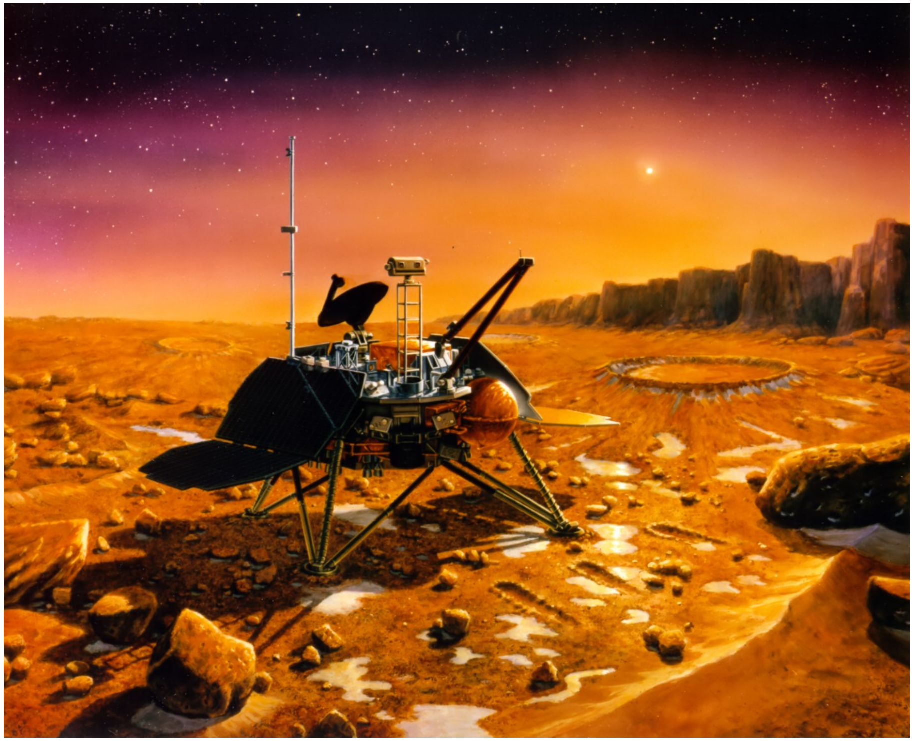
Module 2

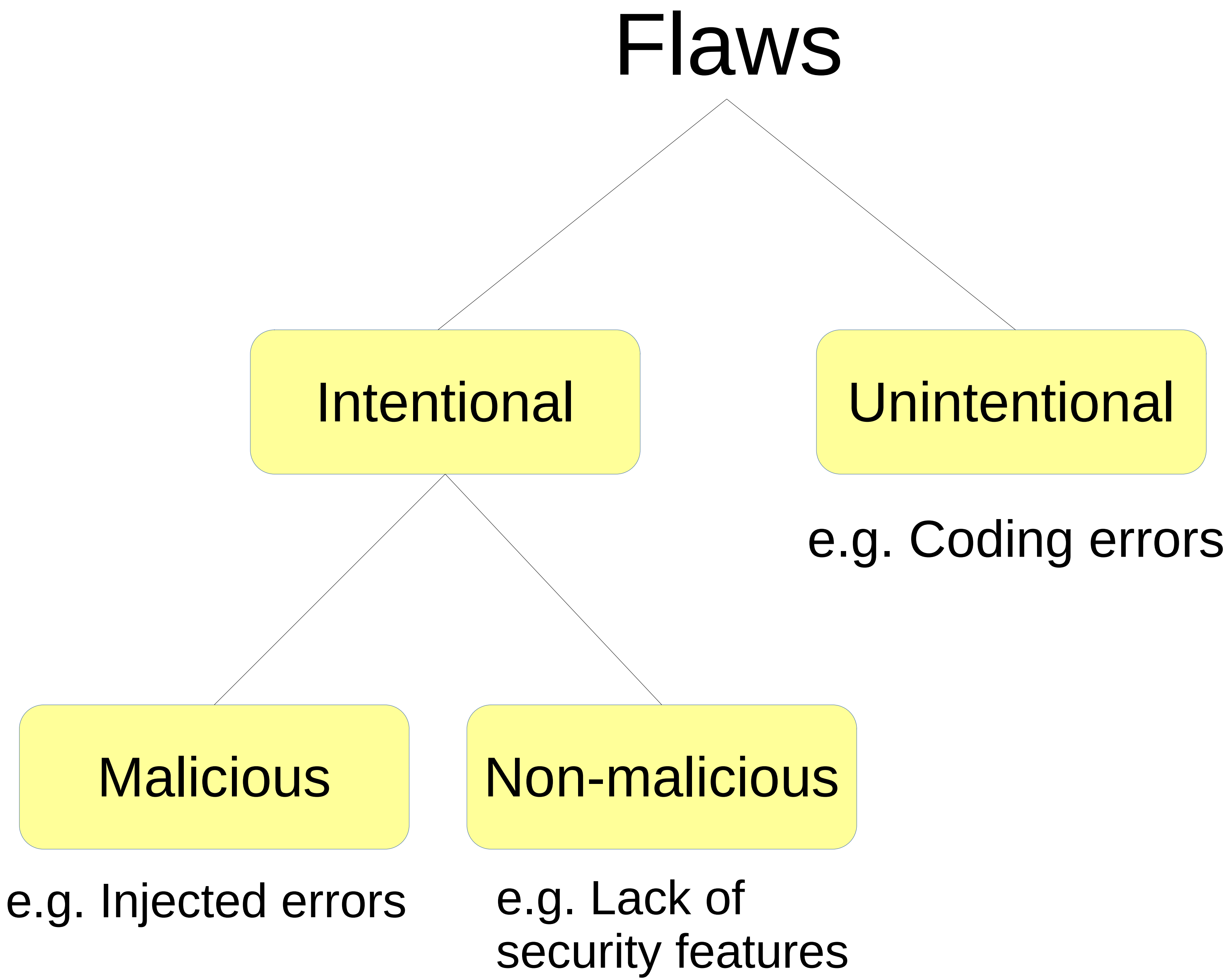
Software Security

Software errors can kill a project



Mars Polar Lander (1999) – crashed on Mars

Sensors were programmed incorrectly and shut off engine;
not caught in testing


Flaws
Intentional
Unintentional
e.g. Coding errors
Malicious
Non-malicious
e.g. Injected errors
e.g. Lack of security features

Unintentional Flaws

We will discuss two types of unintentional flaws:

<u>Local application flaws</u>

- Buffer overread, buffer overflow, TOCTTOU

<u>Web application flaws</u>

- XSS, XSRF, SQL Injection

Buffer overread

Your own memory may look like this:

wake up; have breakfast; need to buy milk; turn off the lights; go to class; that man has a strange shirt; fall asleep; wake up

A web server's memory may look like this:

Bob requests main page; Atta wants reply "Cat"; Li sets password to "sup3rsekr1t"; Kate wants image "derpy_cat"; Poe sets secret key; …

Buffer overread

Bob requests main page; Atta wants reply “Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; ...

Memory

Please reply “Cat” (3 letters).

Cat

Please reply “Cat” (5 letters).

Cat”;

Buffer overread

Bob requests main page; Atta wants reply “Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; ...

Memory

Please reply “Cat” (100 letters).

Cat”; Li sets password to “sup3rsekr1t”; Kate wants image “derpy_cat”; Poe sets secret key; ...

Buffer overread

Supposed to be the size of that array, but user declares this

Heartbleed (2015)

memcpy(bp, pl, payload);

Returned to client

Points to an array

Buffer overflow

Also “stack smashing”, “buffer overrun”

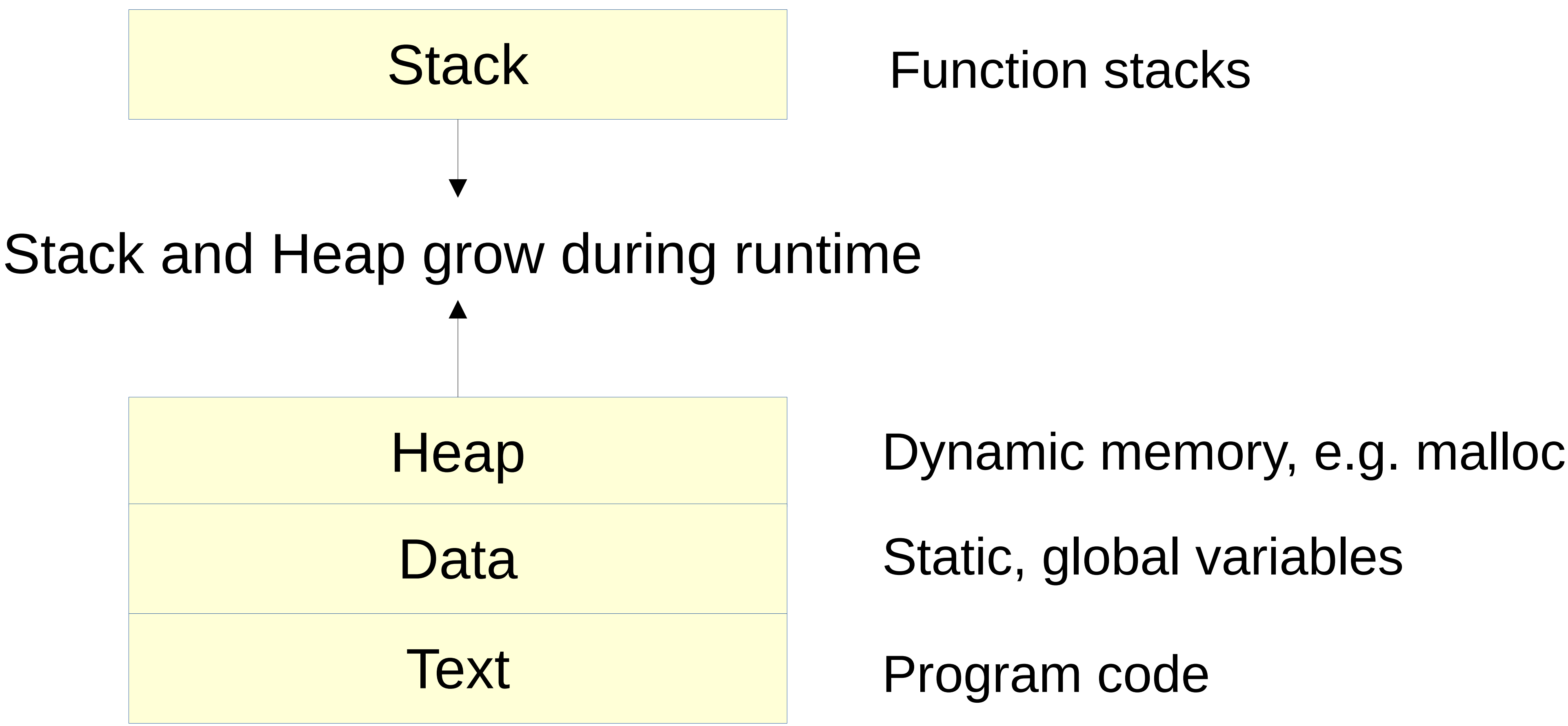
```
void input_username(...) {
   char username[16];
   printf(“Enter username:”);
   gets(username);
   ...
}
```

strcpy, gets, fgets, etc. can write more data than the target size

What if you could write directly into memory?

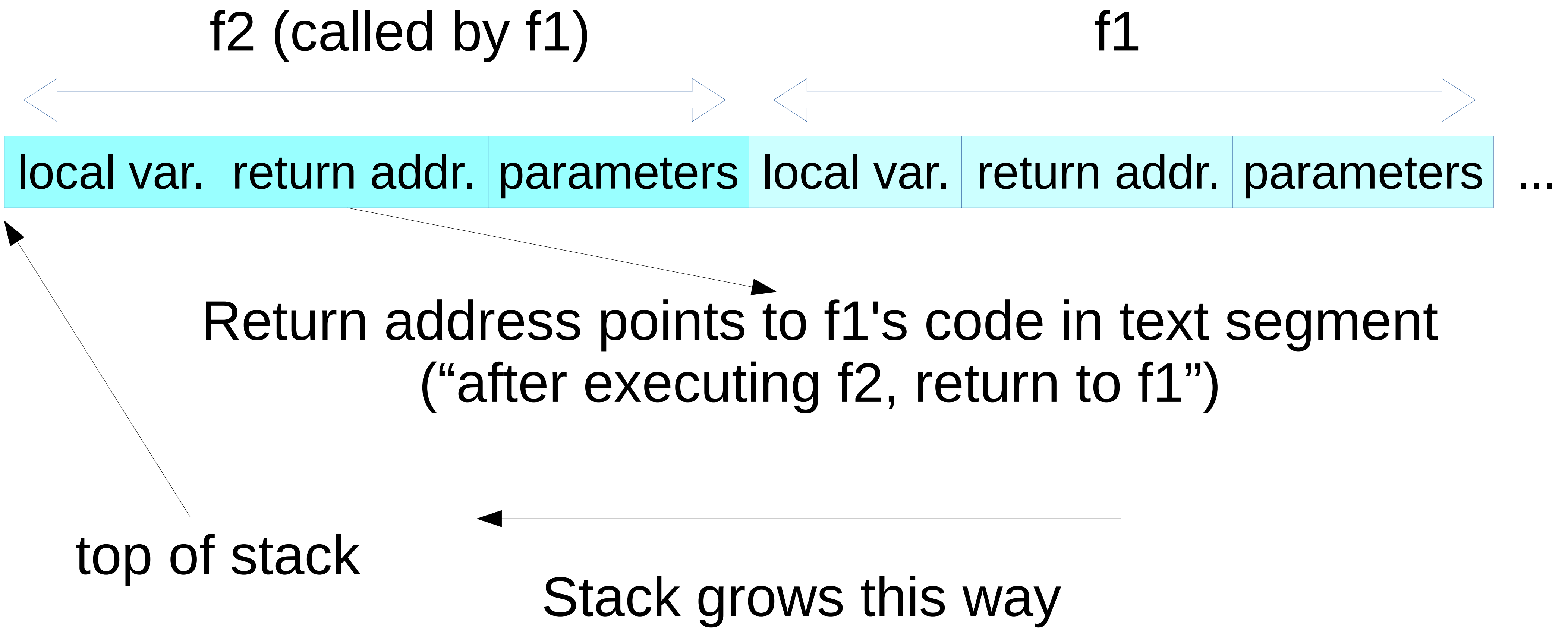
Buffer overflow

Memory of C program process:



Buffer overflow

A simplified function stack



Buffer overflow

A simplified function stack

```
void input_username(...) {
   char username[16];
   printf(“Enter username:”);
   gets(username);
   ...
}
```

gets does not check bounds!

```
[                                   ] [7FA2]
```

...	username[16]	return addr.	Parameters

(return address normally points to text segment, not stack)

Buffer overflow

A simplified function stack

```
void input_username(...) {
   char username[16];
   printf("Enter username:");
   gets(username);
   ...
}
```

If user types 24 A's...

```
[AAAAAAAAAAAAAAAA]   [AAAA]   [AAAA]
```

...	username[16]	return addr.	Parameters

Upon function termination, return to "AAAA" (segfault)

But the attacker can be smarter

Buffer overflow

A simplified function stack

`[execute evil code;]`

another_buffer

Malicious shell code can be written in the stack too

(shell code is assembly code that grants root)

`[AAAAAAAAAAAAAAAA] [E4FF] [AAAA]`

...	username[16]	return addr.	Parameters

This will cause the shell code to be executed!

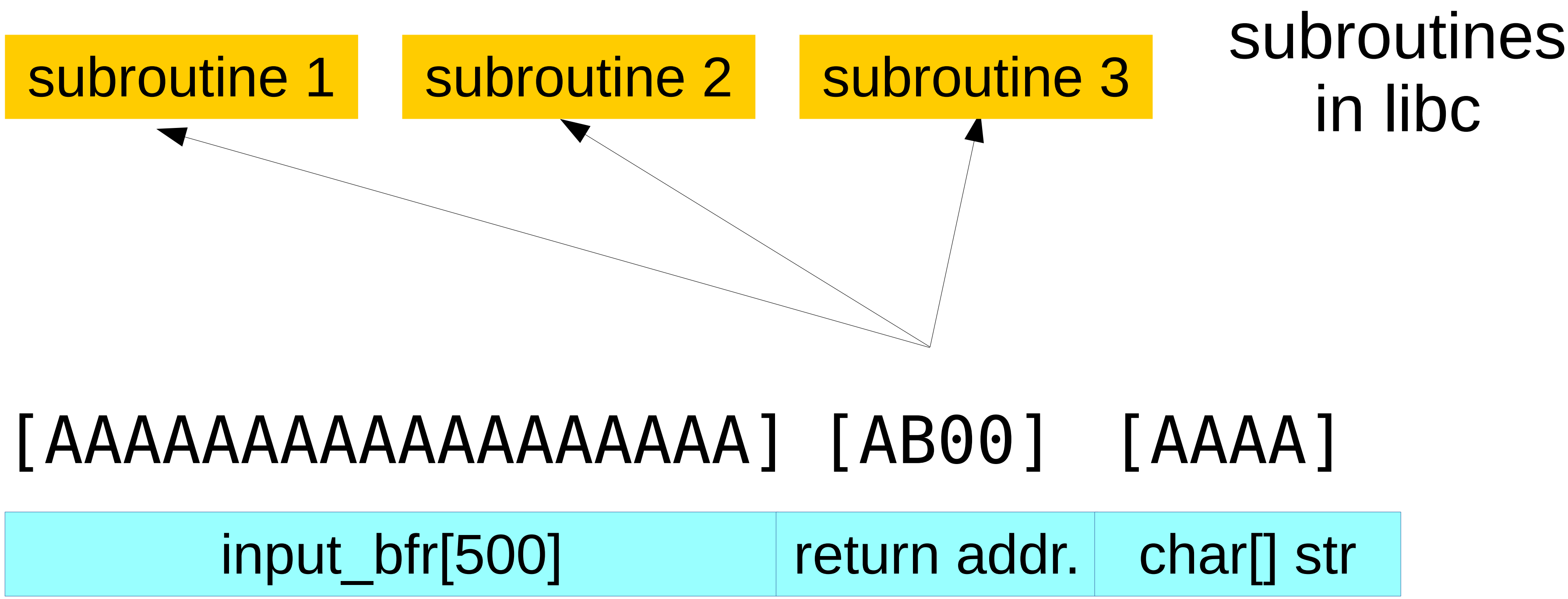
Buffer overflow

Defenses

- Never execute code on stack
  - W^X (write XOR execute), NX, or DEP
- Randomize stack
  - Address Space Layout Randomization
- Detect overflow
  - Canaries
- Don't use C

Buffer overflow

Return-Oriented Programming

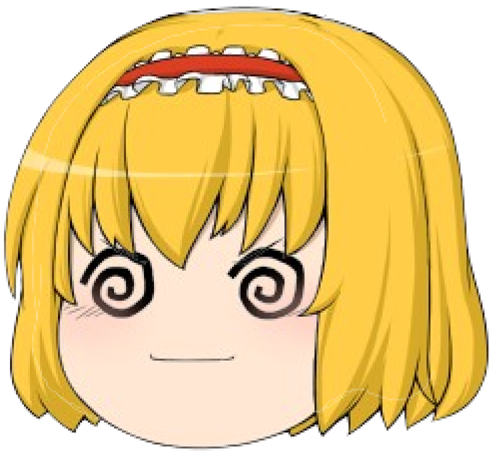


How to defeat W^X

Buffer overflow

Majority of known software flaws are buffer overflows

- Very common (why?)
- Very powerful – gives root access
- Not much harder to exploit than to detect



Integer overflow

- Integers are often stored in 32-bit
  - Sometimes 16-bit with specific systems
- When exceeding the maximum, the result is an error
  - Often, wrapping back to the lowest/negative number
- It is surprisingly easy to exceed the maximum!
  - e.g. What is 2^31 milliseconds?
  - e.g. Any multipliers that can be applied

Format string vulnerability

- The following prints today's lucky number:

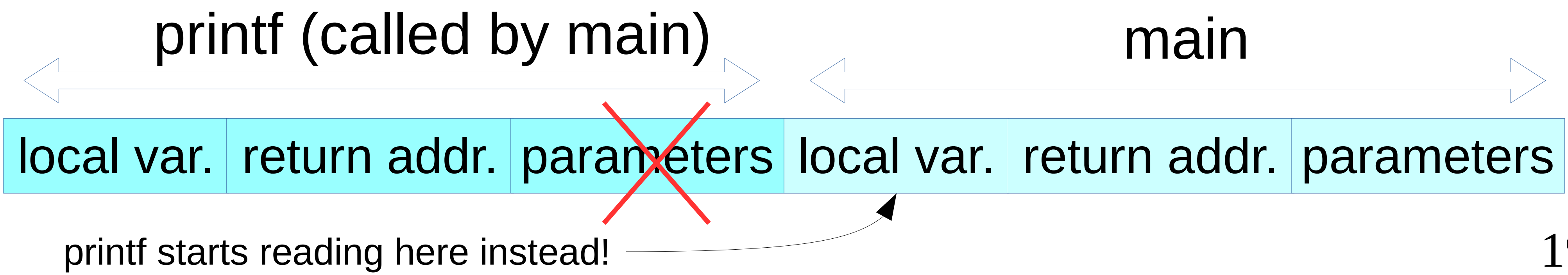
```
printf(“Today's lucky number is %d”, 18);
```

- What about the following?

```
printf(“Today's lucky number is %d”);
```

- What if the user has control over this string?

```
char uname[250];
fgets(uname, 250, stdin);
printf(“Your username is: ”);
printf(uname);
```



Format string vulnerability

- %n: Counts the number of bytes written so far, writes it to the given variable

```
int len;
printf("This string length is%n...? ", &len);
printf("%d", len);
```

```
> This string length is...? 21
```

- What if len was not provided?
- If the user controls a format string, they can put a clever combination of %d and %n there to write whatever they want to an address!

TOCTTOU

A type of "race condition"

- "Time of Check To Time of Use"
- Check: Should the user have privilege?
  - Access control, check ownership, etc.

What if something changes?

- Use: Do something for the privileged user
  - Read file, write to file, change permissions

TOCTTOU

passwd example (pseudocode)

I want to change root password, but I am not root

> passwd new_password

System code:

```
check_access(password_file, user);

update_file(password_file, new_password);
```

What if you can change password_file in-between?

TOCTTOU

passwd example (pseudocode)

> passwd new_password

System code:

attacker: set password_file to point to user_password

```
check_access(password_file, user);
```

attacker: set password_file to point to root_password

```
update_file(password_file, new_password);
```

(Attacker actions are on the OS, not part of the code)

TOCTTOU

Attacker can increase chance of success by:

- Opening a file in a deep directory
- Opening a file in a remote network location
- Simply timing the attack well or keep retrying

Prevention:

- Locking the object under use
- Checking if identifiers have changed later (?)

Cross-site Scripting (XSS)

madeupChat v0.1

Alice [07:31]: hi

Carol [07:33]: yo

Bob [07:34]: hey

EMPEROR [07:55]: `<code>Execute Order 66!</code>`

Alice [07:55]: Yes, Lord!

Bob [07:55]: Yes, Lord!

Carol [07:55]: Yes, Lord!

(example) madeupChat v0.1 has a vulnerability that let chatters execute code on other chatters

Cross-site Scripting (XSS)

XSS vulnerabilities occur when users can write code onto a web page

- Persistent XSS vulnerability
  - User changes content of a page persistently
  - e.g. social media profile page
- Reflected XSS vulnerability
  - Malicious link that executes code as if it was part of the page's content
  - Person who clicks link doesn't know it's evil

www.bad-bank.com/login.php?username=<script>dobadthings</script>

  - e.g. Steal cookies, make fake login window, send messages to other users

Cross-site Request Forgery (XSRF)

In XSRF, a malicious forged link causes the user to make a request that harms herself

Example:

If the victim is currently logged into bad-bank.com:

www.bad-bank.com/give_money.php?amount=10000&target=attacker

SQL injection

Poor SQL code with parsing vulnerability:

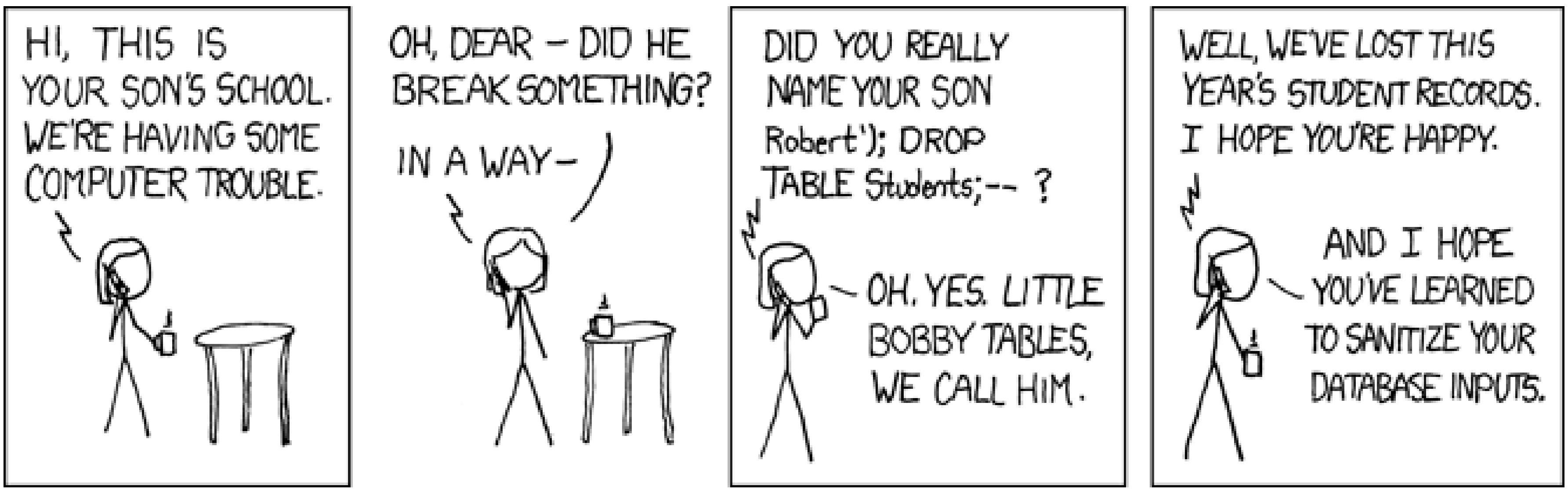
s = “SELECT uid FROM utable WHERE username =‘” + input_uname + “’AND password =‘” + input_password + “’”

If uid is non-empty, then login is successful.

User inputs input_uname as:

‘ OR ‘1’ = ‘1'--

SQL injection



Parsing vulnerabilties

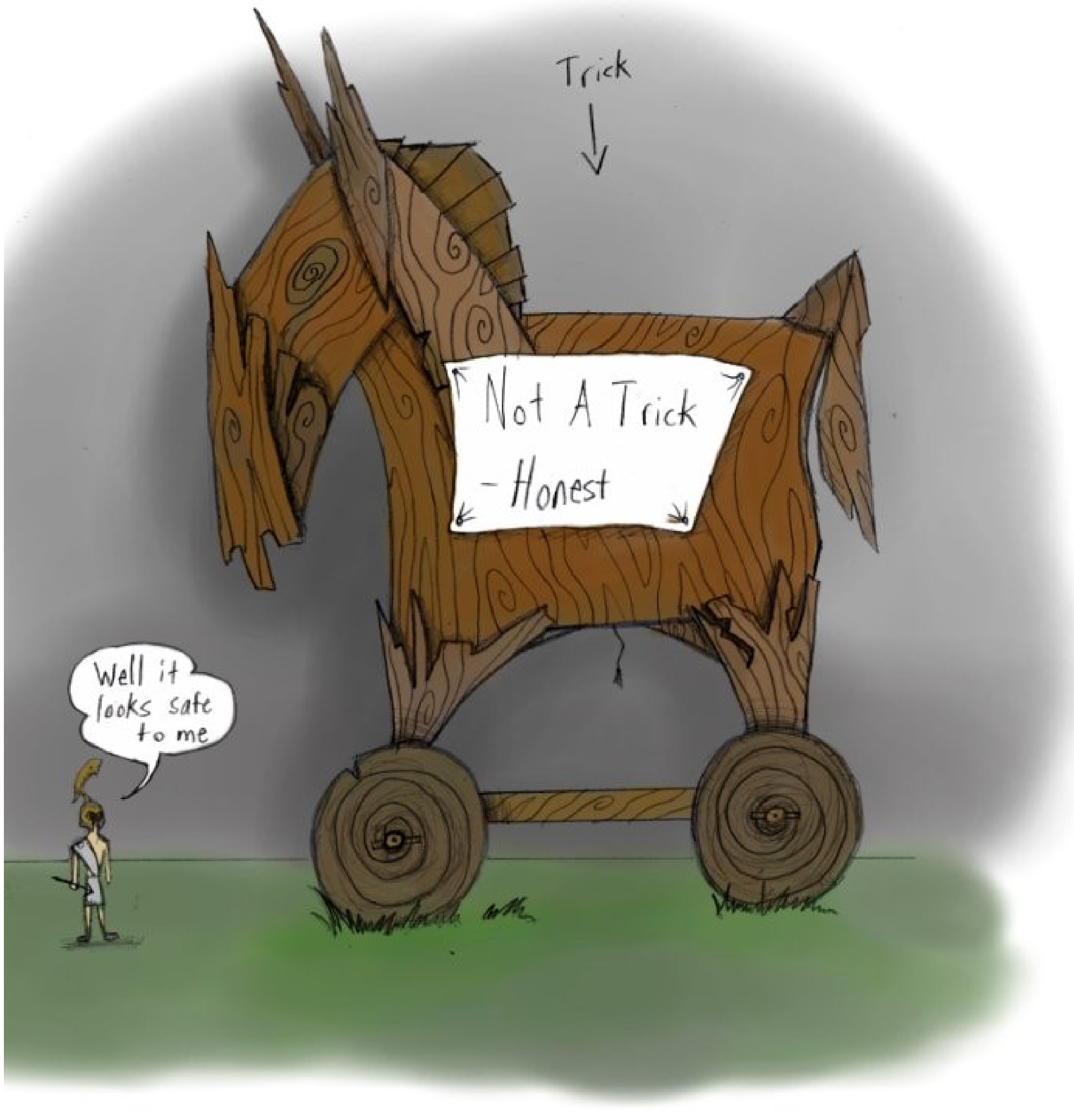
Characters and numbers may be parsed incorrectly:

- rlogin -l -froot attack allowed remote login as root
  - Target computer receives “login -f root”
- Canonicalization: Many ways to represent the same string; attacker chooses a way to avoid blocking/detection. Examples:
  - http://2130706433/
  - A trojan downloading a file with .exe%20 to avoid exe files being blocked
  - System allows access to /data/user/taowang, so you access data/user/taowang/../../../system/

Classifying malware

- Malware consists of a *spreading mechanism* and a *payload*
- We can classify by method of spread
  - AKA infection vector
  - How does it get on your computer?
- Or by effect on system (payload)
  - What does it do to your computer?

Trojan



Trojan

"Given a choice between dancing pigs and security, users will pick dancing pigs every time."

–Gary McGraw and Edward Felten, "Securing Java"

Trojan

A trojan is a piece of malware that spreads by tricking the user into activating/clicking it

- Packaged with useful software
- Looks like useful software (e.g. Android re-packaging)
- Scareware
- Spear phishing

> *People often represent the weakest link in the security chain.*

— Bruce Schneier

Trojan

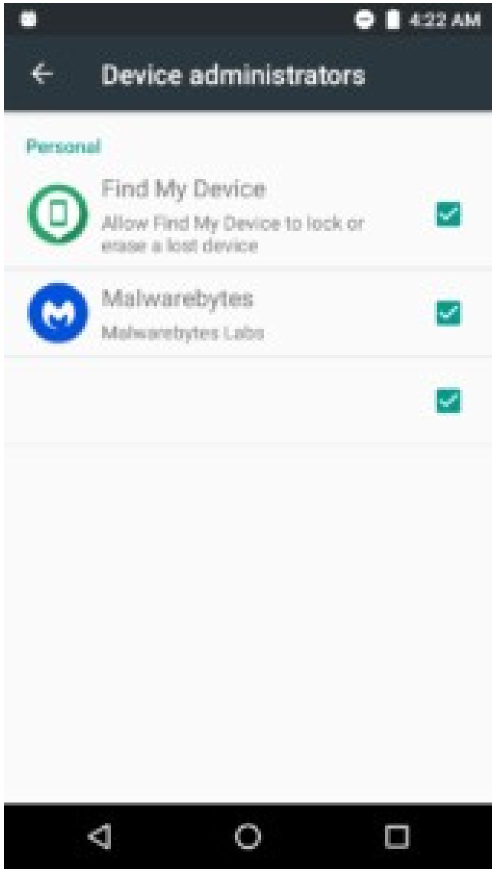
ILOVEYOU (2000, Windows):

- Malware in e-mail attachment: “LOVE-LETTER-FOR-YOU.txt.vbs”
- Destroys files on target system through replication
- Reads mailing list, sends files to them
- Downloads another trojan “WIN-BUGSFIX.EXE”
- Very easy to reprogram

Trojan

Conficker Worm’s interface illusion

Trojan



MobiDash’s interface illusion

Removable media

ByteBandit (1987, Amiga):

- Spreads with an infected floppy disk
- Resides in memory, even after reboot
- Infects all inserted floppy disks
- After causing 6 infections, black screen!

Network

Malware that spreads through packets requires *no user action*

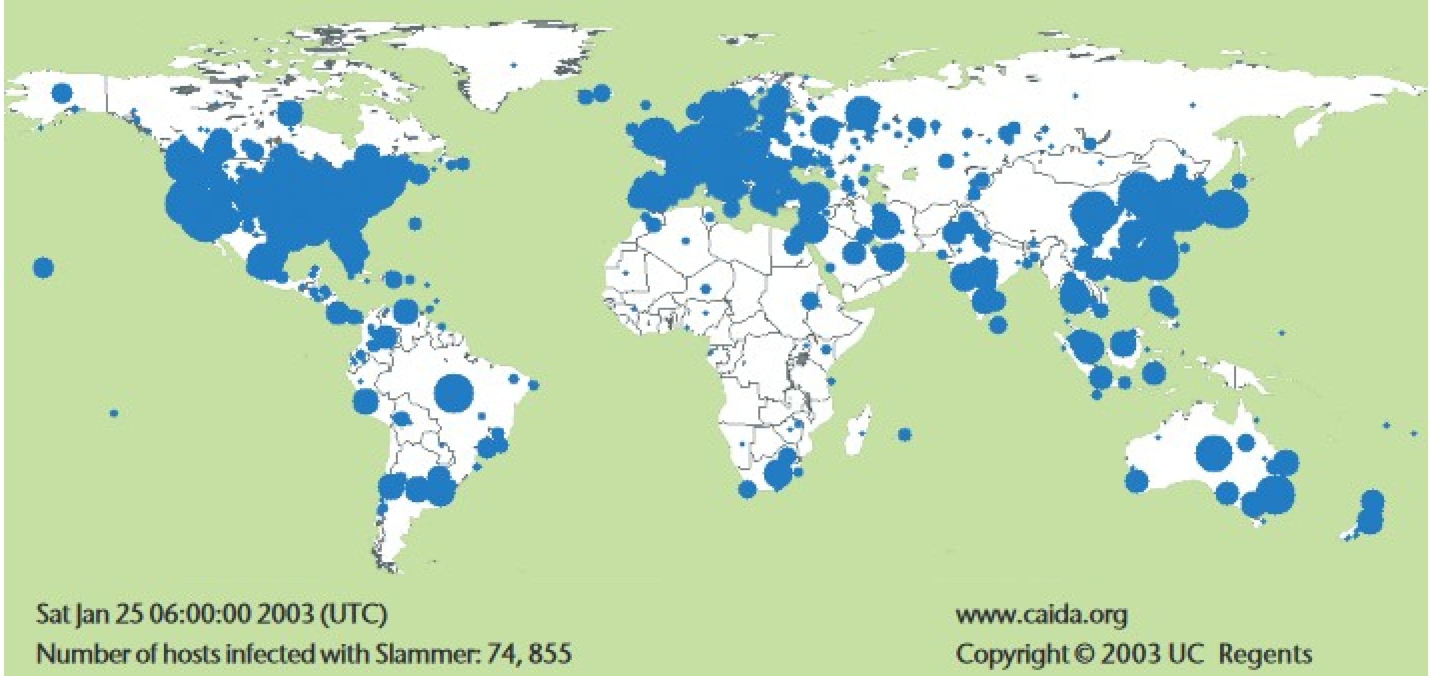
- Infects network-facing background programs (daemons) to spread
- Can be very fast – infection and spread can be automatic, exponential
- Malware spreading explosively can cause worldwide internet outage, and are called “worms”

Network

Slammer Worm (2003, Microsoft SQL Server):

- Exploits SQL Server buffer overflow using a packet
- Patch had existed after Blackhat warning
- Generate random addresses, sends itself by UDP
- Infection doubled every 8.5 seconds, reached 90% of all vulnerable systems in 10 minutes
- “Warhol worm” - Andy Warhol “In the future, everyone will be world-famous for 15 minutes”
- No payload

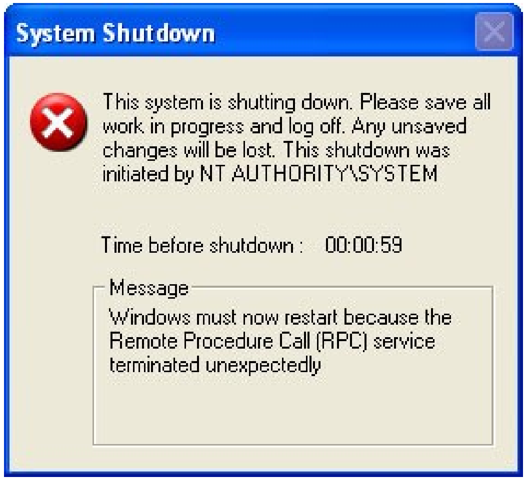
Network


Sat Jan 25 06:00:00 2003 (UTC)
Number of hosts infected with Slammer: 74, 855
www.caida.org
Copyright © 2003 UC Regents

Network

Blaster Worm (2003, Windows):

- Exploits RPC buffer overflow
- Payload: DDoS windows update site
- Earlier warnings, patches were not installed
- (Unintentionally) shut down computers
- Welchia is a “helpful” worm that removes Blaster and force-installs patches



Planted malware

Installed intentionally by an attacker who has (temporary) control over the system:

- Employee
- Espionage
- From other malware

Sometimes the payload is a logic bomb: Malicious code set off by specific conditions

- After some amount of time
- If an employee is fired

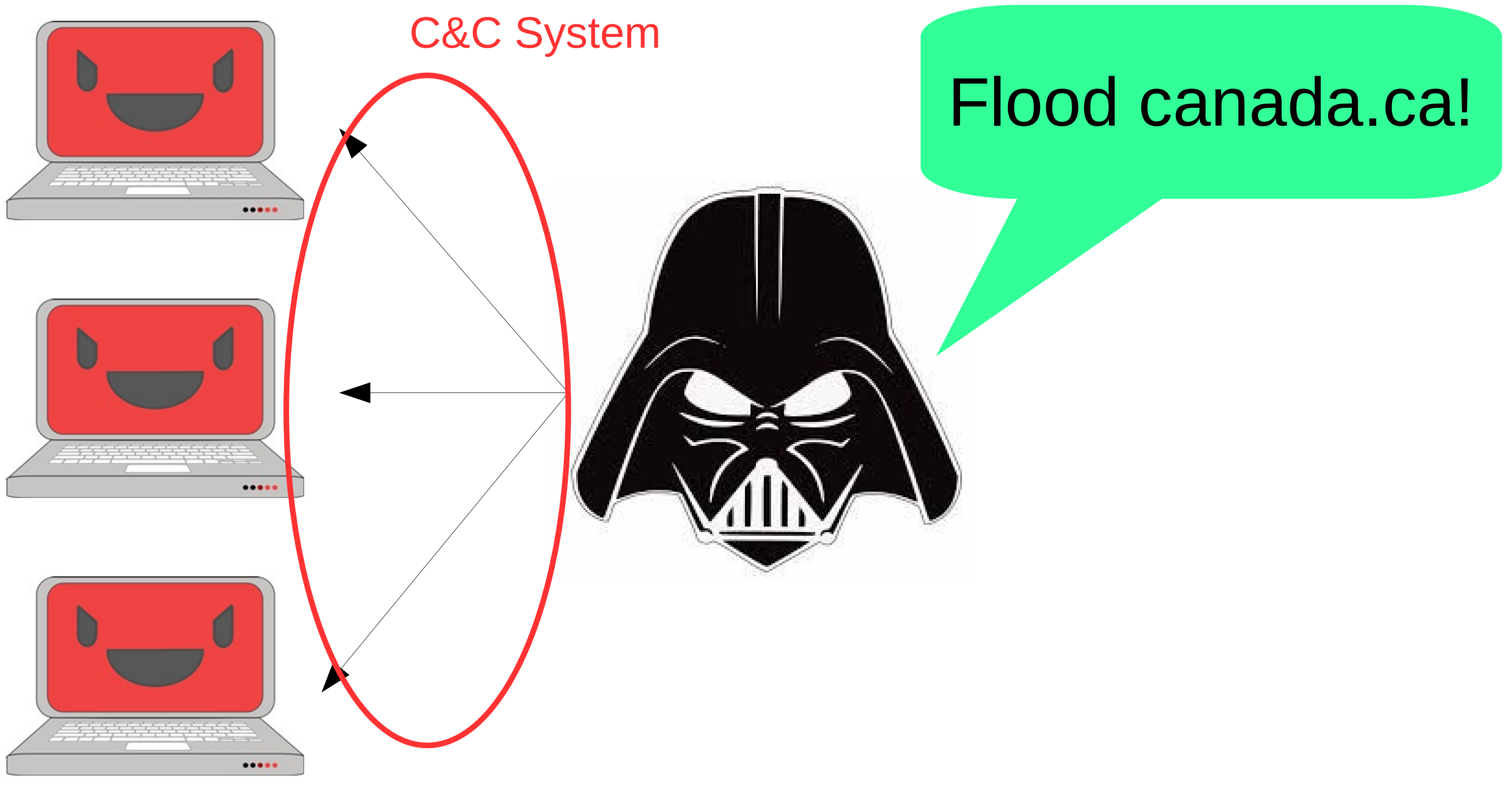
Classifying malware

- Malware consists of a *spreading mechanism* and a *payload*

- We can classify by method of spread
  - AKA infection vector
  - How does it get on your computer?

- Or by effect on system (payload)
  - What does it do to your computer?

Botnet



Computers owned by different users

Botnet

- Consists of three components:
  - A Master
  - A large number of infected devices (“bots”)
  - A Command and Control structure
- Useful for:
  - Hiding attack source/identity
  - Sybil attacks
  - Malware spreading
  - Spam

Backdoors

- Allows unexpected access to system
- Could be created on system because:
  - Left for testing (intentional non-malicious flaw)
  - Installed by malware
  - Demanded by law

Rootkits

- A rootkit is a piece of malware for maintaining command & control over a target system (root)
- It changes the behavior of system functionalities to hide itself/some other malware
- Hard to remove
- User rootkits can change files, programs, libraries, etc.
- Kernel rootkits can change system calls

Rootkits

Sony XCP (2005)

- **Rootkit** by Sony
- Garbles write-output of XCP disk
- Hides all files and folders starting with “sys”
- Eventually, Sony released an uninstaller due to pressure

Zip bombs, compiler bombs

- Destructive payloads usually used in the context of a trojan
- Zip bombs: Unzipping the bomb creates a very large file
- Compiler bombs: Compiling the bomb creates a very large file
- Besides destruction, can be used to break certain scans

Spyware


zuck
Instagram HQ
500M likes
zuck Thanks to everyone in our community for helping us reach this milestone!
kevin @zuck

Spyware

- Secretly collects data about the user

Pegasus (2016):

- Spyware for iOS and Android
- Developed by software company NSO Group
- Reads text messages, traces the phone, can enable microphone and camera, etc.
- Uses three zero-days, including Use After Free

Trackers (Spyware)

- **Cookies** store information about you
- Third-party cookies allow your actions on site A to be collected and sent to site B (blocked on some browsers)
- Web beacons on websites make a request for you to a third-party (ad) server, which can also automatically send your cookies for that server
- Beacons in multiple sites often link to the same ad server

```
<img src="http://ad.doubleclick.net/ad/pixel.quicken/NEW" width=1 height=1 border=0>
```

Keylogging

Several kinds of keyloggers:
- Application-specific keyloggers
- Software keyloggers
- Hardware keyloggers

Each can be installed covertly

Some keylogging malware steals your credentials (e.g. “bankers”)

Ransomware

CryptoLocker: Estimated $3 million extorted

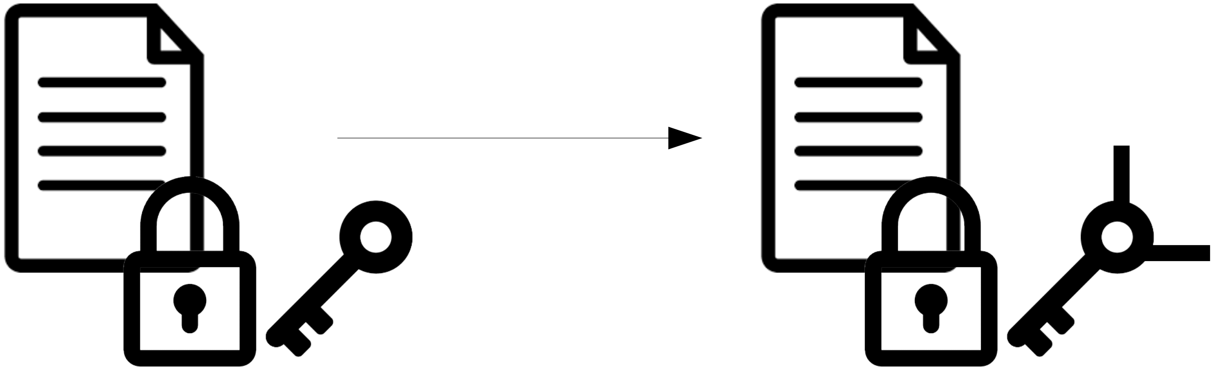
Ransomware

- General technique: encrypt disk, then demand ransom to decrypt it
- Disk is encrypted using public key, private key is on attacker's own server
- Attached storage media will also be encrypted
- Little recourse once files are encrypted
- A number of attacks fail to release keys

Stealth techniques

To avoid detection:

- Polymorphic code
- Hide in memory, disguise file patterns
- Interrupt scanning techniques



Code polymorphism

Advanced Persistent Threats

- Combination of multiple infection vectors and spreading strategies
- Focused, long-duration attack
- Achieves political/industiral goal

Advanced Persistent Threats

Stuxnet (2011)

- Spreads by network and USB
- Uses four zero-day attacks
- Does nothing in almost any machine
- But it wrecks a specific type of Iranian nuclear reactor centrifuge controller
- Speculated to be government-sponsored

Advanced Persistent Threats

Advanced Persistent Threats

Flame (2012)

- Spyware: records keystrokes, camera, screen, sends to remote server
- Behavior determined by your antivirus
- Uses a fake certificate obtained by attacking a Microsoft server's weak cryptorgaphy
- Very large (20MB)
- Attempted to erase itself when discovered

Covert Channels

Covert channels are resources (not intended for communication) that are used by an attacker to communicate information in a monitored environment *without alerting the victim*

- To retrieve stolen data
- To receive commands
- To update malware

Examples: TCP initial sequence number, size of packets, timing, port knocking

Side Channels

Side channels leak information in unintended ways

- Power analysis
- Timing analysis
- EM wave analysis
- Acoustic analysis

Defenses: air gap, Faraday cage, etc.

Side Channels

Spectre (2017)

Side channel attack on microprocessors

1) CPU branch prediction can be trained by attacker-controlled data

2) A branch mis-prediction can read process memory and affect processor cache

3) Processor cache contents can be exposed using timing attacks

=> This can potentially leak any process memory

Side Channels

Spectre (2017)

Example (Kocher et al.):

```
1    if (x < array1_size)
2        y = array2[array1[x] * 4096];
```

- The attacker can make the CPU “expect” that the check in line 1 will pass, and predictively execute line 2
- If the CPU runs line 2 on x larger than array1_size, it is a buffer overread
- This affects the processor cache and what it reads can be guessed with a timing attack

Defensive strategy

How do we defend against software flaws?

- Blocking access from attackers: Scanning, ...
- Writing good code: code review, change management, testing
- Fixing bad code: code analysis, patching

Malware scanning

- Signature-based:
  - Scans for virus “signatures”
  - Scans memory, registry, program code
- Behavior-based (“heuristics”):
  - Detects system irregularities
  - May have false positives
- Sandboxing
  - Run potentially malicious code in controlled environment
  - Often used with honeypots

Code analysis

Look for vulnerabilities/bugs in code

- Static code analysis

  Examine code for vulnerabilities

- Dynamic code analysis

  Test code by running it on input

- Formal verification

  Prove that code follows a specification

Code analysis

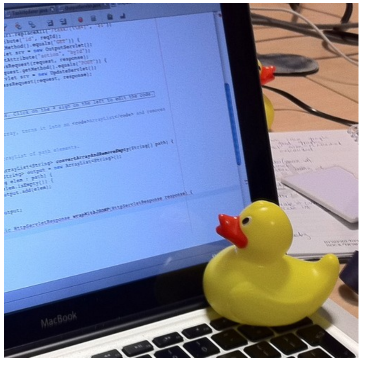
sel4: Formally verified OS

- Contains 8,700 lines of C, 600 lines of assembly
- Proof of correctness: 200,000 lines of code
- Can have “unintended features”
- Bugs that are not in the specification could still exist (e.g. timing attacks)

Software testing

- Unit testing (test small units one at a time)
- Integration testing (test integration of units)
- Fuzz testing (test with random input)
- Black-box testing (test unknown system)
- White-box testing (test known system)
- Regression testing (test if update causes bugs)

Code review



- Formal inspection
  - Programmer explains code to panel
- Pair programming
  - Programmer explains code to an observer
- Rubber duck programming
  - Programmer explains code to themselves
- Change management
  - System for recording and managing code changes

Patching

Error 503 Service Unavailable

Service Unavailable

Guru Meditation:

XID: 1995750753

Varnish

Having a good error message helps!

Patching

Several unresolved problems:

- Vulnerable users don’t install patches
- Patches cause further issues
- Patches don’t resolve underlying issues

Microsoft’s “Patch Tuesday” forces patches to be installed and makes it easier for system administrators to fix issues

Summary

Unintentional flaws

- Buffer overread, buffer overflow, TOCTTOU
- XSS, XSRF
- Exploited by malware: viruses, worms, trojans

Intentional malicious flaws

- Planted malware, rootkits

Intentional non-malicious flaws

- Covert channels, side channels

Defensive strategy

- Scanning, code analysis, testing, review, patching

Module 3

Internet Security and Privacy

Some communication mediums are unsafe

What can be eavesdropped upon?

- Air (for broadcast messages such as wireless)
- Copper wires (vampire tap)
- Optical fiber 光纤
- Devices (phones, computers, etc.)

<u>Our goals:</u>

- **Confidentiality** – Safeguard packets from eavesdropping
- **Integrity** – Prevent packet modification in transmission
- **Authenticity** – Prove the identity of the sender

Cryptography

A cryptosystem consists of:

- Key(s)
- Encryption mechanism
- Decryption mechanism

Kerckhoffs' Principle states that:

The key(s) of a cryptosystem should be hidden,
but the mechanisms should be public.

(Why?)

The XOR function $\oplus$

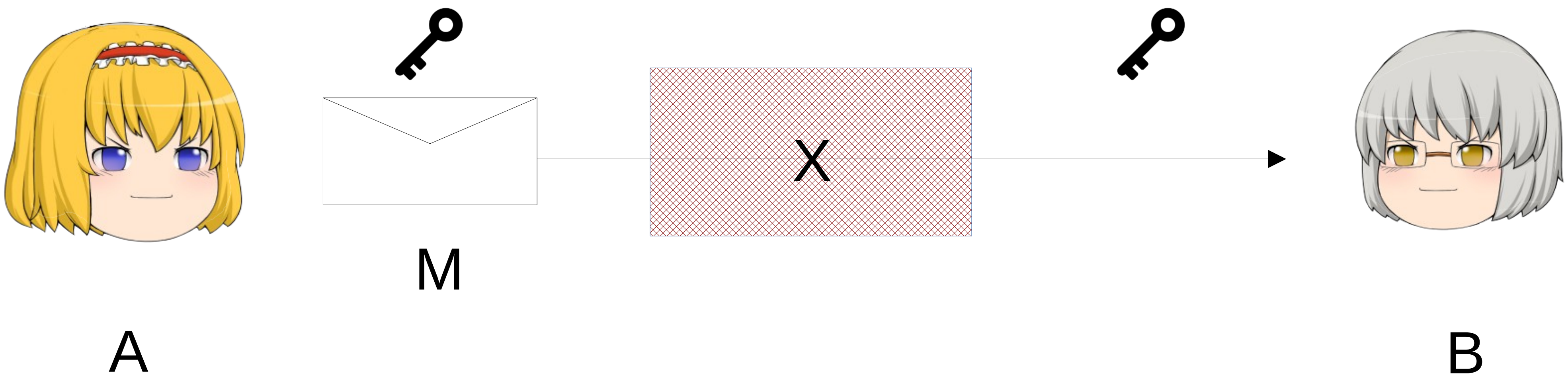
Value table of XOR:

$\oplus$	**0**	**1**
0	0	1
1	1	0

XOR is the same as "Addition modulo 2".
Bit-by-bit XOR of two bit strings:

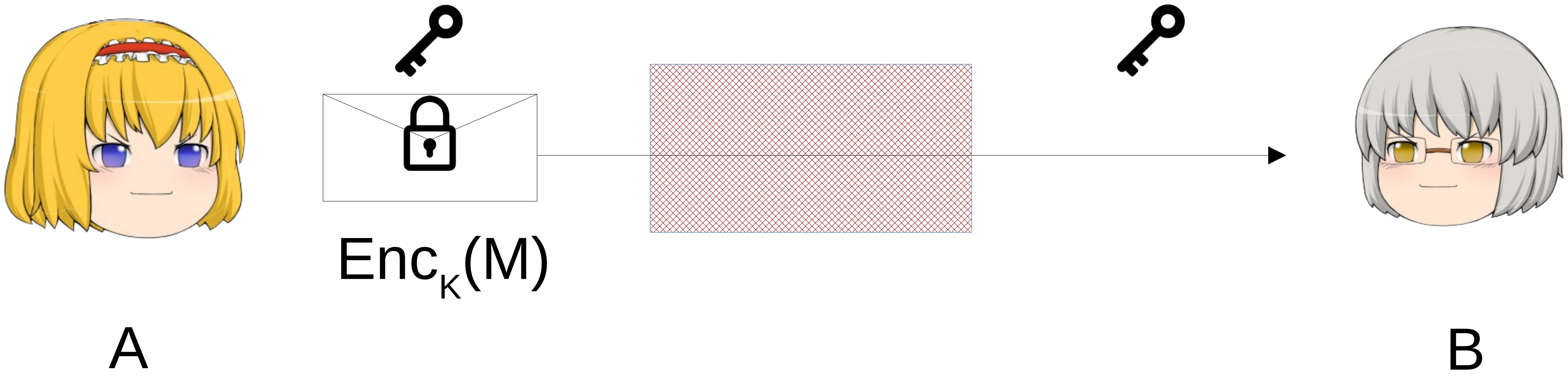
$$(0110) \oplus (1011) = (1101)$$

Encryption and decryption



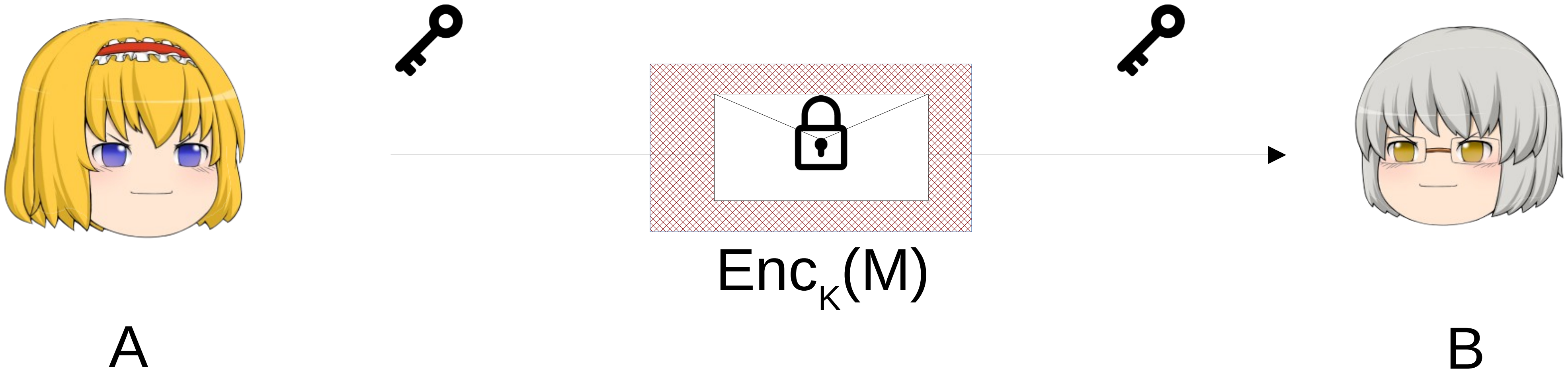
Scenario: A wants to send **plaintext** M to B, but doesn't want the attacker to see M when it passes through the unsafe medium (red).
A and B both already know some key K.

Encryption and decryption



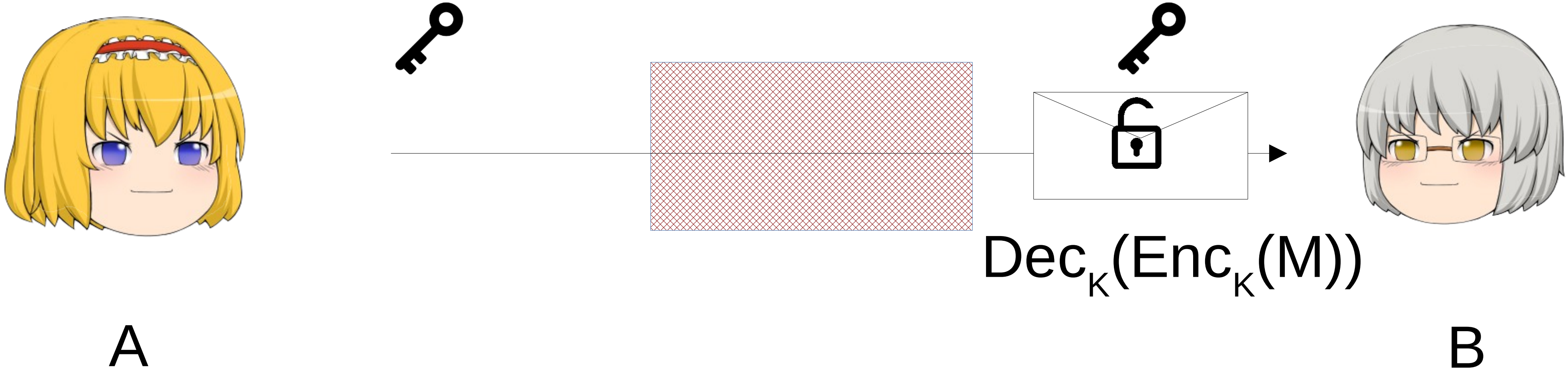
1. Using the encryption mechanism Enc() and key K, A encrypts M to a **ciphertext**, $Enc_K(M)$.

Encryption and decryption



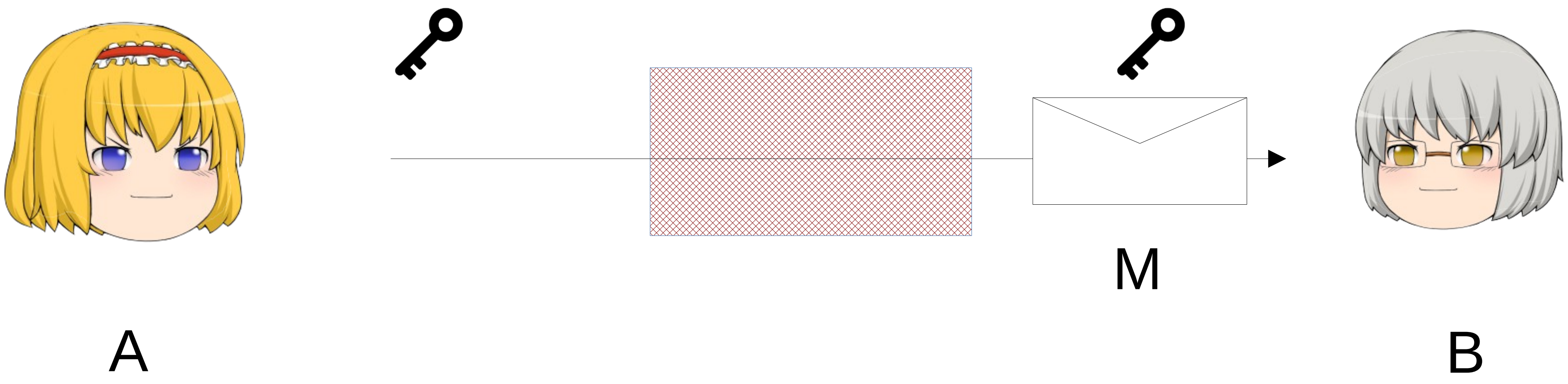
2. A sends $Enc_K(M)$ across the channel.

Encryption and decryption



3. B receives $Enc_K(M)$, and decrypts it using the decryption procedure Dec() and key K.

Encryption and decryption



4. Dec(Enc(M)) = M; B receives the plaintext message M.

Simple System: The Caesar Cipher

Plaintext: `ATTACK “LONG A”`

Encryption: Add K letters (K=1 here)

Ciphertext: `BUUBDL “MPOH B”`

Unsafe medium

Ciphertext: `BUUBDL “MPOH B”`

Decryption: Subtract K letters (K=1 here)

Plaintext: `ATTACK “LONG A”`

Simple System: The Caesar Cipher

Problems of this cryptosystem:

- **Ciphertext Repetition**: What if you see `BUUBDL “MPOH B” and then EFGFOE “MPOH B”`?
- **Key Update:** For security, we should update the key frequently. How can we do so?
- **Short Key Length:** How many possibilities are there for the encryption/decryption mechanism?
- **Frequency analysis:** If the letter “F” appears most frequently in ciphertexts, what does it mean?

Solving the Ciphertext Repetition Problem

Use a Initialization Vector (IV):

- The IV “modifies” the key for encryption

$$Enc_{K,\ IV}(M)$$

- Each message must have a different IV

  -> Even with the same key and plaintext, a different IV will produce a different ciphertext

- The IV is sent **publicly** alongside the message – it does not matter if the attacker sees it

Solving the Ciphertext Repetition Problem

Plaintext: `ATTACK “LONG A”`

Encryption: Add K+3 letters (K=1)

Ciphertext: `EXXEGO “PSRK E”, +3`

IV

Unsafe medium

Ciphertext: `EXXEGO “PSRK E”, +3`

Decryption: Subtract K+3 letters (K=1)

Plaintext: `ATTACK “LONG A”`

Solving the Key Update Problem

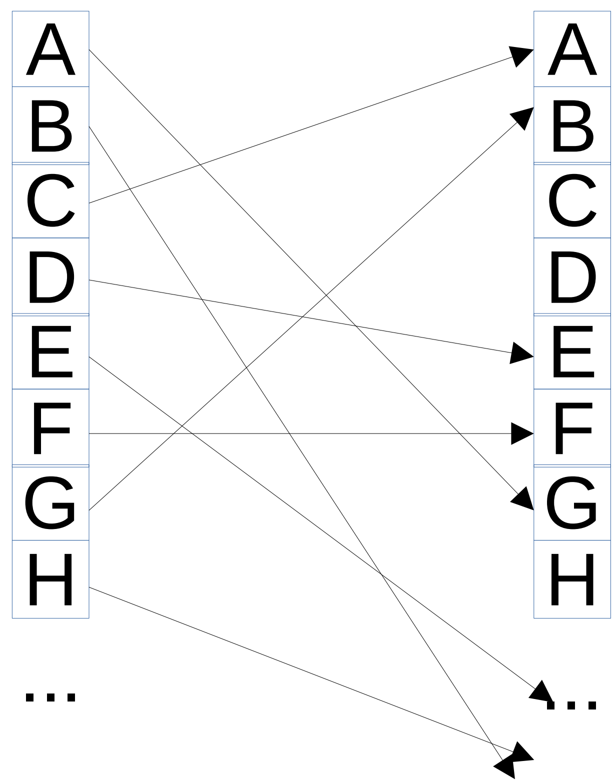
Find a safe channel to deliver the key instead

- Hand-delivered documents, cards
- Not practical for computer systems

Public Key Encryption

- In PKE, the encryption and decryption keys are different
- This can be used to create a safe channel on an unsafe one
- Only send the encryption key across the channel
- More later

Solving the Key Length Problem (Substitution Cipher)



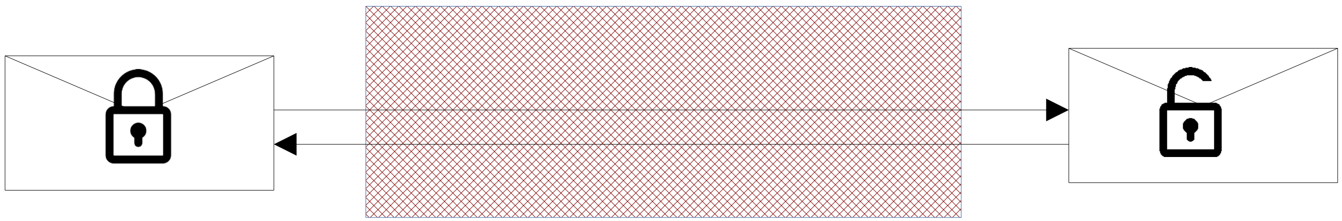
Plaintext Ciphertext

How many variations are there?
26! ~= 2^{88} => Key length is “88 bits”

Solving the Key Length Problem (Substitution Cipher)

We will use
variation number
309273 to converse.

Sent in safe channel



$\mathrm{Enc}_{309273}(M)$ $\mathrm{Dec}_{309273}(M)$

The “variation number” is the cryptosystem's **key**

Solving the Frequency Analysis Problem

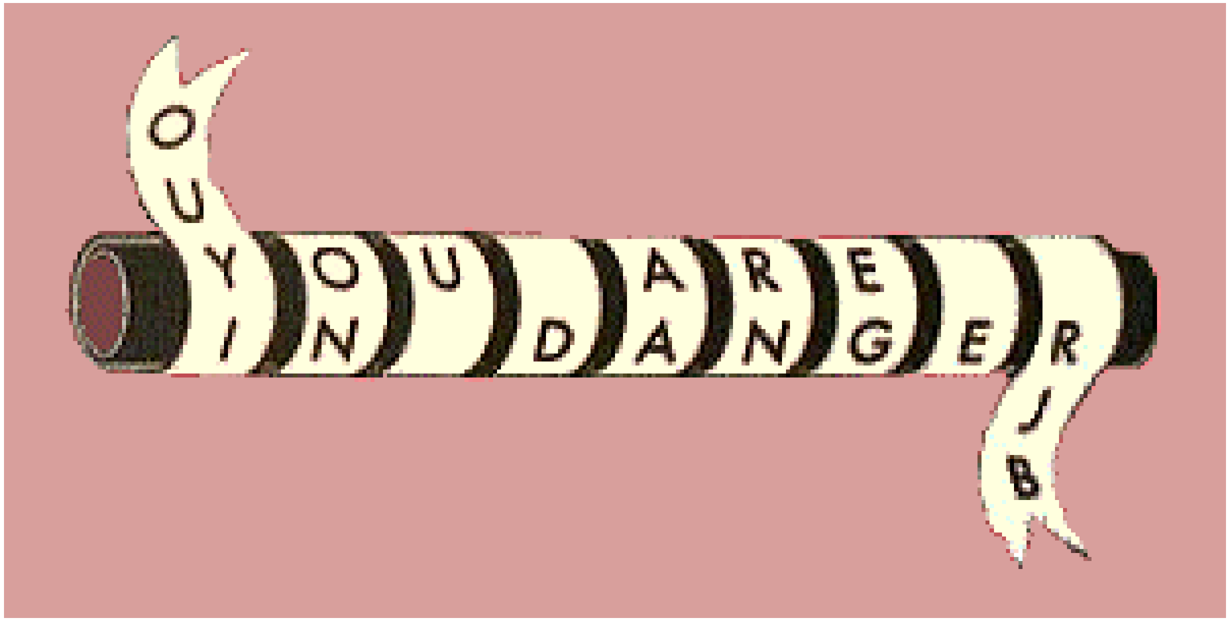
- We cannot do this easily – all substitution ciphers are weak to frequency analysis (cryptograms!)
- One suggested solution (Vignere ciphers): shift different letters based on their position using a key
  - e.g. key = DOG (4 15 7), then shift 1^{st} letter by 4, 2^{nd} by 15, 3^{rd} by 7, 4^{th} by 4, 5^{th} by 15, ...
  - Easily defeated! (How?) 有足够时间，会找到cycle并且破解
- Broader category of cryptanalysis can defeat almost all “homemade” cryptography

Symmetric Key Encryption (SKE)

- A type of cryptosystem where *the two parties both know a secret key.*
- If the key is K, then the encryption and decryption algorithms are $Enc_K()$ and $Dec_K()$.
- $Enc_K()$ and $Dec_K()$ are public, but K must be secret.
- $Enc_K(M)$ should not reveal either K or M.
- Both parties can encrypt and decrypt.

We will discuss three types: OTP, Stream Ciphers and Block Ciphers

Scytale



What is the key in this cryptosystem?

Enigma machine

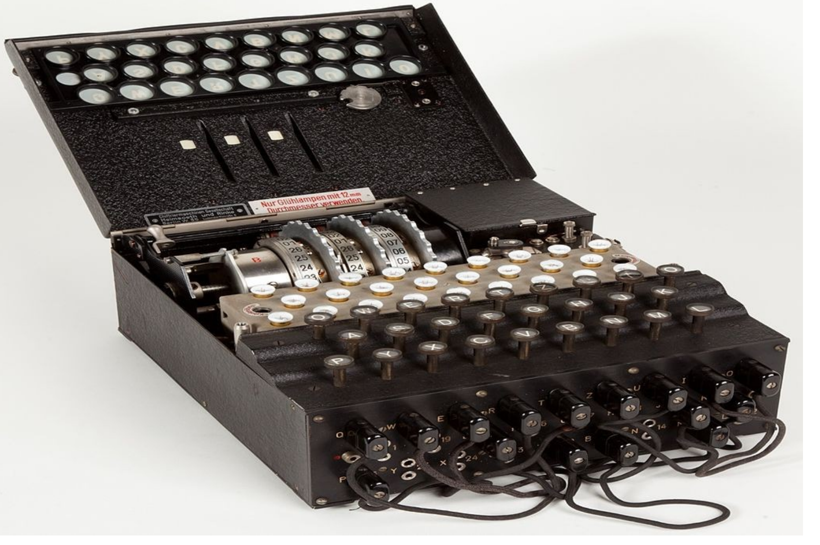
- The key is the rotor position
- Codebook contains an initial position

1)Set to initial position

2)Type a new position

3)Set machine to new position

4)Type message



One-Time Pad

Plaintext: Write in bit form (e.g. “ABC”)

01000001 01000010 01000011

Key: Uniformly random bit sequence

10110100 01010101 10001111

Encrypt: Bit-by-bit XOR key with plaintext

Ciphertext: 11110101 00010111 11001100

Decrypt: Bit-by-bit XOR key with ciphertext

01000001 01000010 01000011

One-Time Pad

“Perfectly” information secure if:

- Key is truly uniformly random
- Key is only used once, ever

(Why is it perfectly secure?)

VENONA project code-breakers (1943)

One-Time Pad

<u>Breaking a Two-Time Pad:</u>

Suppose the attacker intercepts two ciphertexts:

$$C = M \oplus K \text{ and } C' = M' \oplus K$$

The attacker applies XOR to the ciphertexts to obtain:

$$\begin{aligned} C \oplus C' &= M \oplus K \oplus M' \oplus K \\ &= M \oplus M' \end{aligned}$$

The result is the XOR of the plaintexts.

- If the attacker correctly guesses M, he can obtain M' by $M \oplus C \oplus C'$.
- If the attacker correctly guesses only one word of M (and its position), he can still obtain some letters in M' (at the same position) – he can drag this guess around and observe the result, known as crib-dragging.

Stream cipher

Generates keystream of any length
from **random seed**

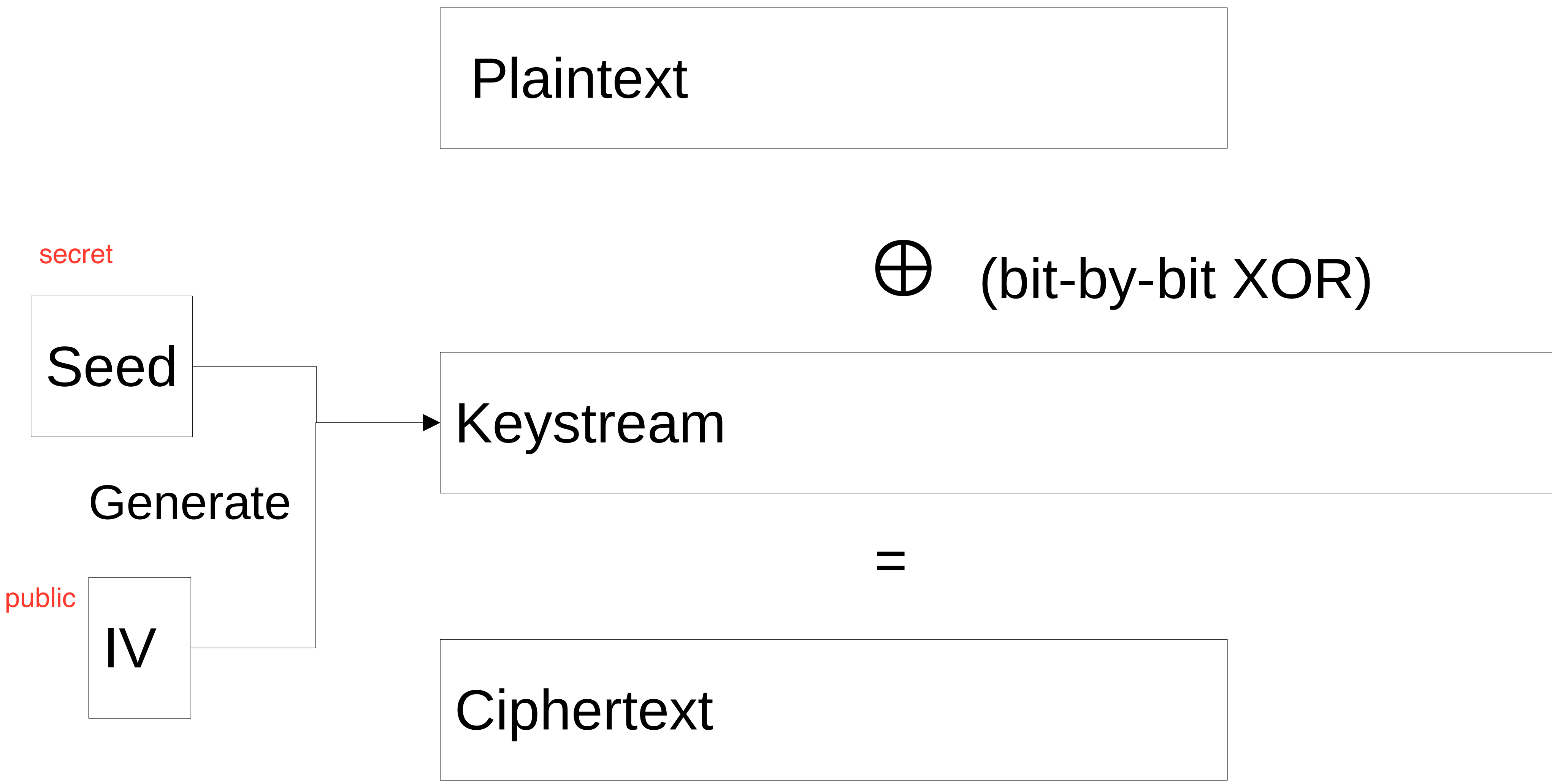
- Keystream is pseudorandom
- ~~Key is truly uniformly random~~
- Seed and IV are only used once, ever

can reuse the seed but the combination of seed and IV are only once

$$\text{Enc}_{\text{seed, IV}}(M) = \text{Keystream}_{\text{seed, IV}} \oplus M$$

Currently used: A5/1 (cell phones), Salsa20 (TLS)

Stream cipher (Enc/Dec)

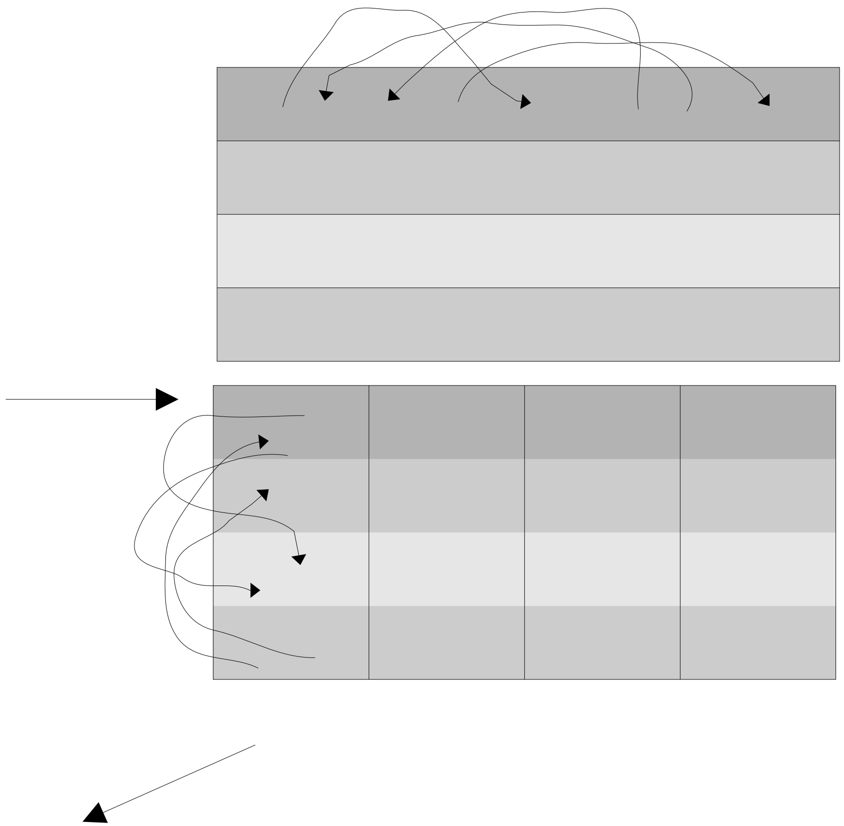


Salsa20 example

Place seed, IV, and position in a
16-by-16 matrix
Each entry is 4 bytes

“expa”	Seed	Seed	Seed
Seed	*“nd 3”*	IV	IV
Position	Position	*“2-by”*	Seed
Seed	Seed	Seed	*“te k”*

Alternatively, scramble each
row and scramble each
column (10 times each)



Output all bits as keystream

Block cipher

Difference from stream ciphers:

- There is a fixed block size (128 bits for AES)
- Plaintext is divided into blocks of this size
- We encrypt each block to produce ciphertext
- The “same” key is used for each block

We must change something,
or we run into the ciphertext repetition problem!

Block cipher

We use the **mode** to avoid the ciphertext repetition problem between blocks:

- <u>Electronic codebook (ECB):</u>

  All keys are the same (no defense against ciphertext repetition)

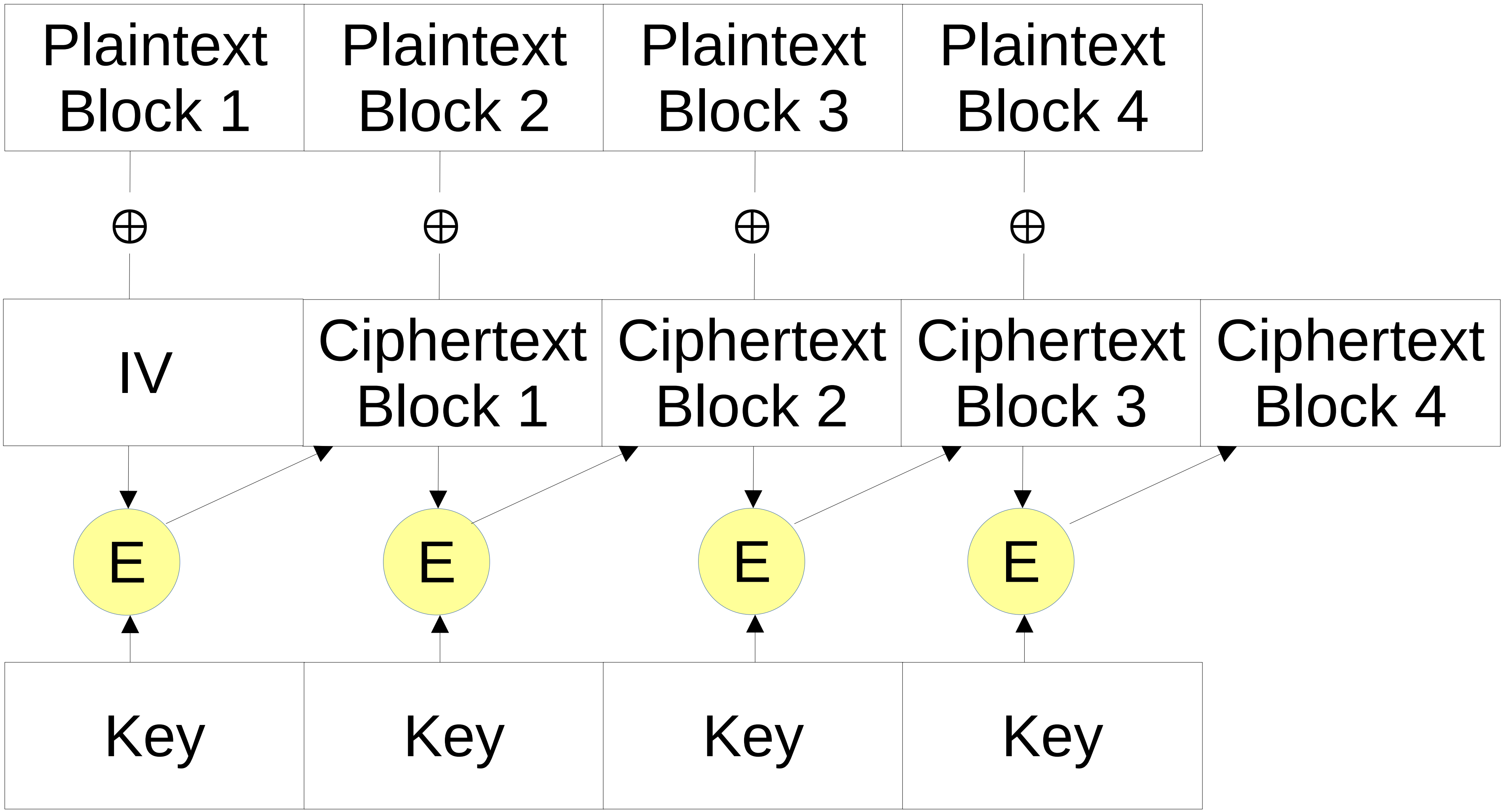
- <u>Cipher Block Chaining (CBC):</u>

  Plaintext block X is XOR'd with Ciphertext block (X-1) before encryption

- <u>Counter (CTR):</u>

  Plaintext block X is XOR’d with a “keyblock” X, generated by an encryption of counter X with an IV

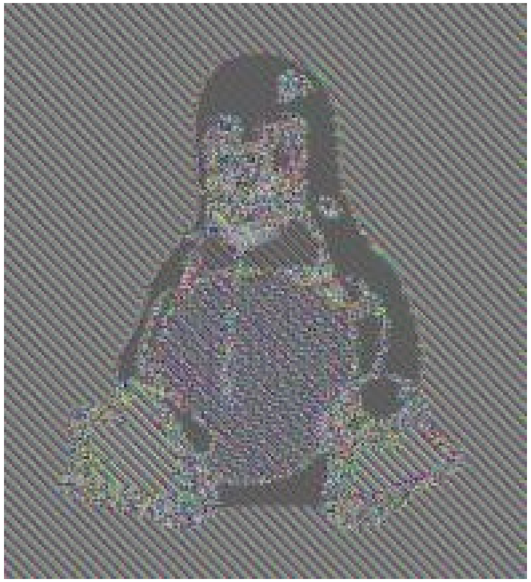
CBC mode (AES):



E is the 128-bit encryption mechanism

Block cipher

Plaintext

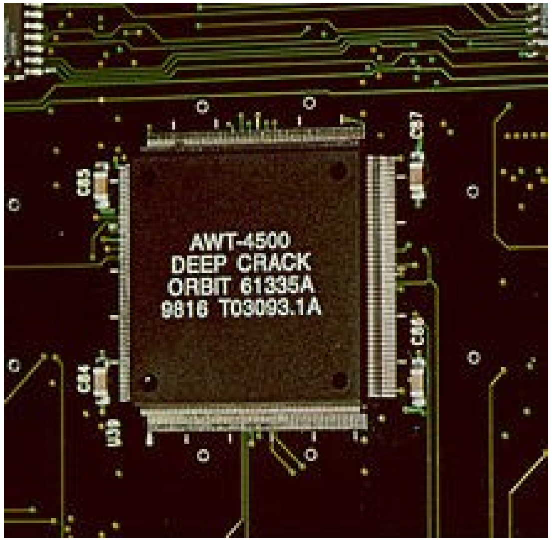


ECB mode

ECB mode is insecure!

Block cipher

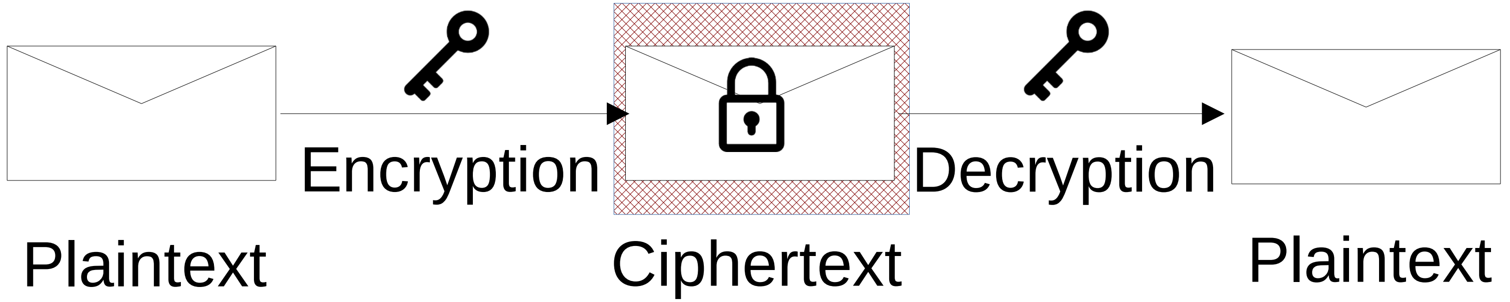
- Includes DES (56-bit), AES (128-bit)
- DES was shown to be too weak in 1998
- AES is the current standard; widely used
- Stream ciphers are generally faster (and keystream can be generated ahead of time)



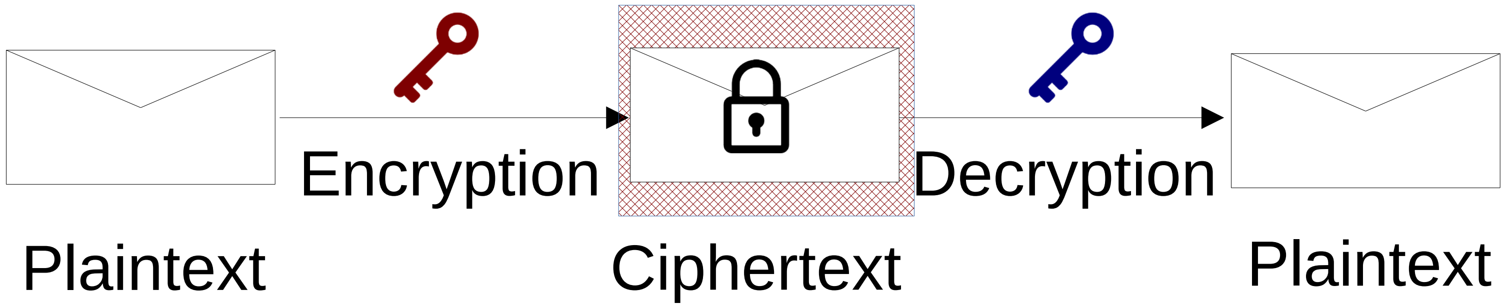
“Deep crack” DES cracker

Public Key Encryption (PKE)

In SKE, locking and opening require the *same key*



What if we want them to require *different keys*?



This is known as *Public Key Encryption*

Public Key Encryption (PKE)

Has two keys for two procedures:

Public key is used for encryption

Private key is used for decryption

Alice generates both keys.

(They are mathematically related.)

Then, Alice publishes her public key:

Anyone can encrypt

Only Alice can decrypt

Anyone can write a message that only Alice can read.

Examples: RSA, ElGamal, ECC

RSA

- First PKE (1977), widely used now in encryption
- Requires much longer keys (2048/4096 bits)
- Less efficient than SKE
- No “perfect security”; can be broken by quantum computers

PKE and SKE

	PKE	SKE
Key	Two: public/private	One: secret
Key setup	Share public key	Need safe channel
Encrypt	Anyone	Both participants
Decrypt	Only key generator	Both participants
Efficiency	Costly to encrypt/decrypt	Cheap

We can combine PKE and SKE to cover their weaknesses

Key Exchange
(using PKE)

1. Alice generates a public/private key pair
2. Alice shares the public encryption key
3. Bob generates a secret key,
   encrypts it with PKE , and sends it to Alice
4. Alice decrypts the secret key,
   and uses it for SKE from now on

What if the private key is leaked?

In practice, the public/private key pair is **short-lived** to guranatee **forward secrecy**

Key Establishment (using Diffie-Hellman)

1. Alice and Bob use some g and prime p, where g generates integers modulo p
2. Alice generates and sends $g^A \bmod p$
3. Bob generates and sends $g^B \bmod p$
4. Alice and Bob compute secret key $g^{AB} \bmod p$
   Alice: $(g^B \bmod p)^A = g^{AB} \bmod p$
   Bob: $(g^A \bmod p)^B = g^{AB} \bmod p$

Other cryptographic tools

We may also want **integrity** and **authenticity**

- Confidentiality: The message is secret
- Integrity: The message is correct
- Authenticity: The sender/receiver's identity is correct

For this, we need other tools:

- Cryptographic hash
- Message Authentication Code (MAC)
- Digital signature

Cryptographic Hash

Cryptographic hashes are irreversible *one-way functions:*

MESSAGE $\xrightarrow{\text{Hash}}$ b194 d920

Properties:

- Output is small, fixed size
- Different inputs may give same output
- Function is publicly known

Examples: MD5 (insecure!), SHA1, SHA2, SHA3

Cryptographic Hash

Cryptograhic hashes need to be difficult for the attacker to reverse or manipulate:

1) Given the output, it is hard to find an input hashing to that output

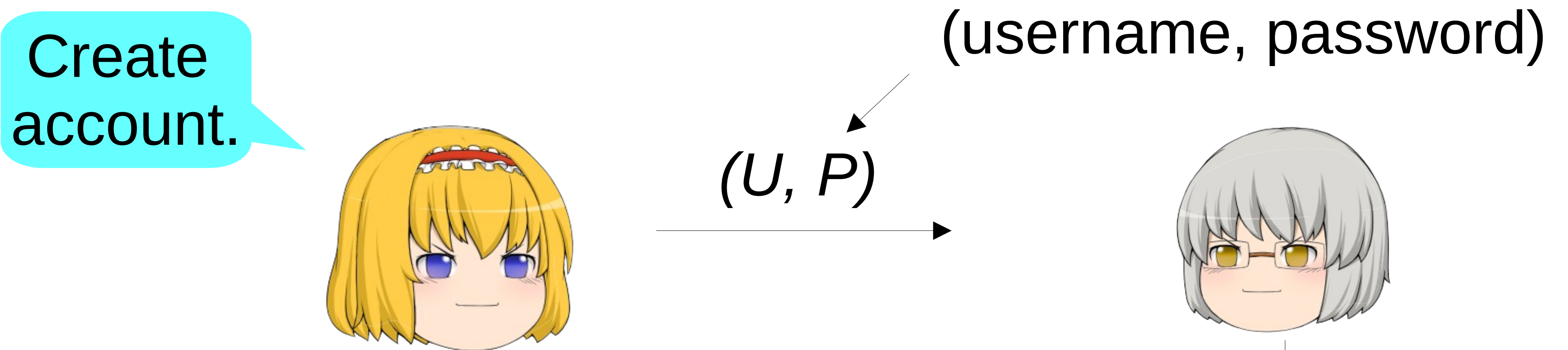
$$\text{????} \xrightarrow{\text{Hash}} \text{5e88 4898}$$

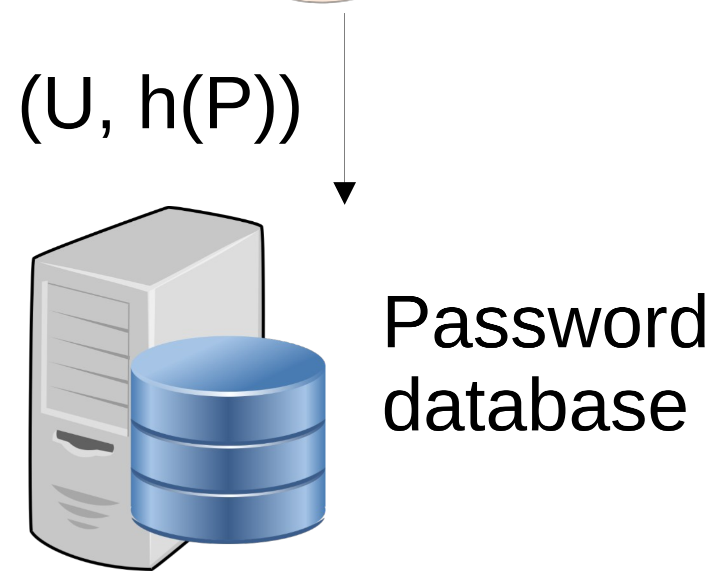
2) A small input change should produce an unpredictable output change

$$\text{MESSAGE} \xrightarrow{\text{Hash}} \text{b194 d920}$$

$$\text{MESSAGF} \xrightarrow{\text{Hash}} \text{e460 d5cf}$$

Cryptographic Hash Password Storage





P is never stored directly or encrypted because the password database can be stolen (even the key!)

Instead, it is hashed for storage

What if two users have the same password?

Cryptographic Hash Password Storage

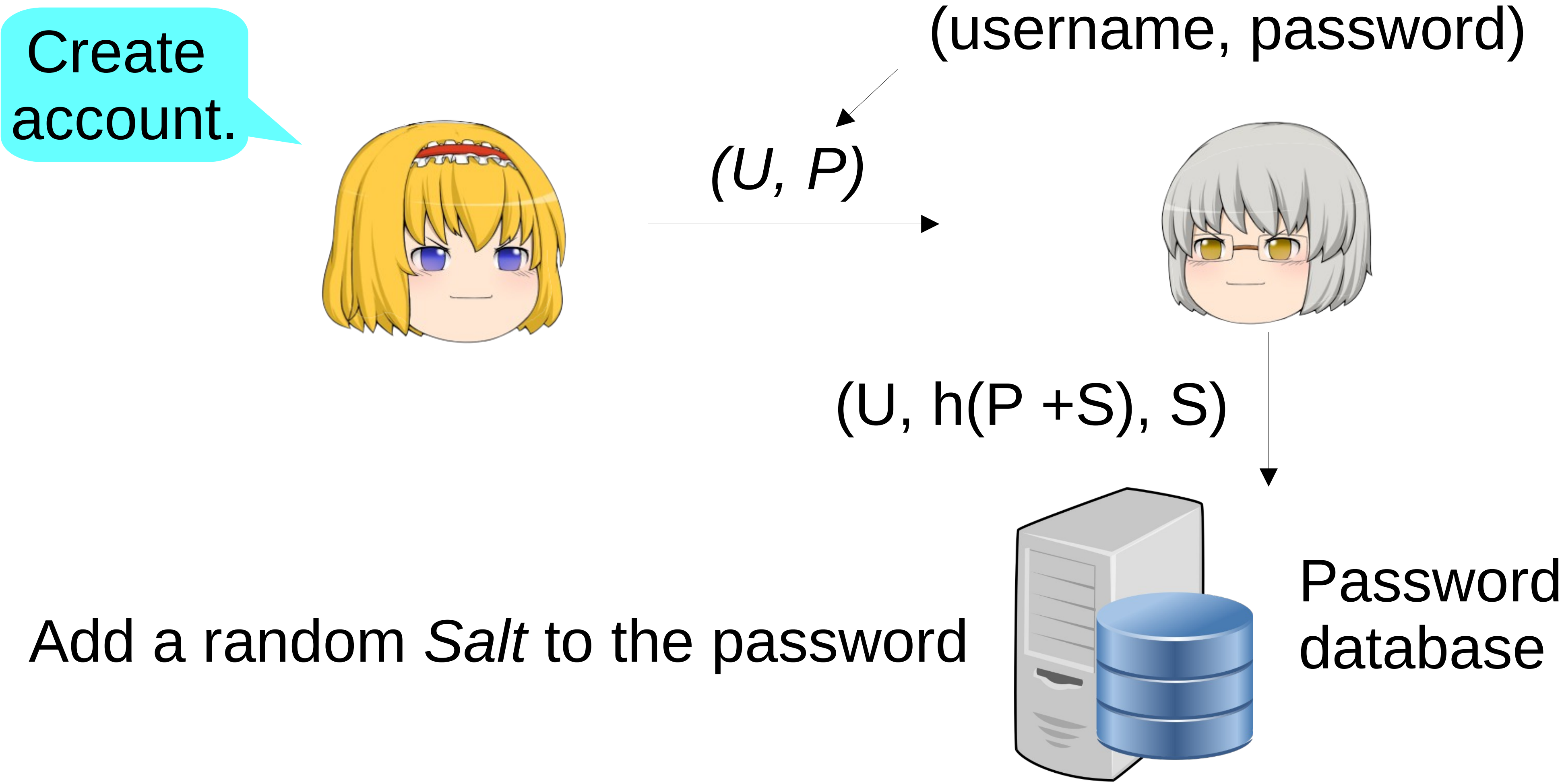
Attacker can *precompute* a hash table:

Guess password	Hash
123456	h(123456)
abc	h(abc)
...	...

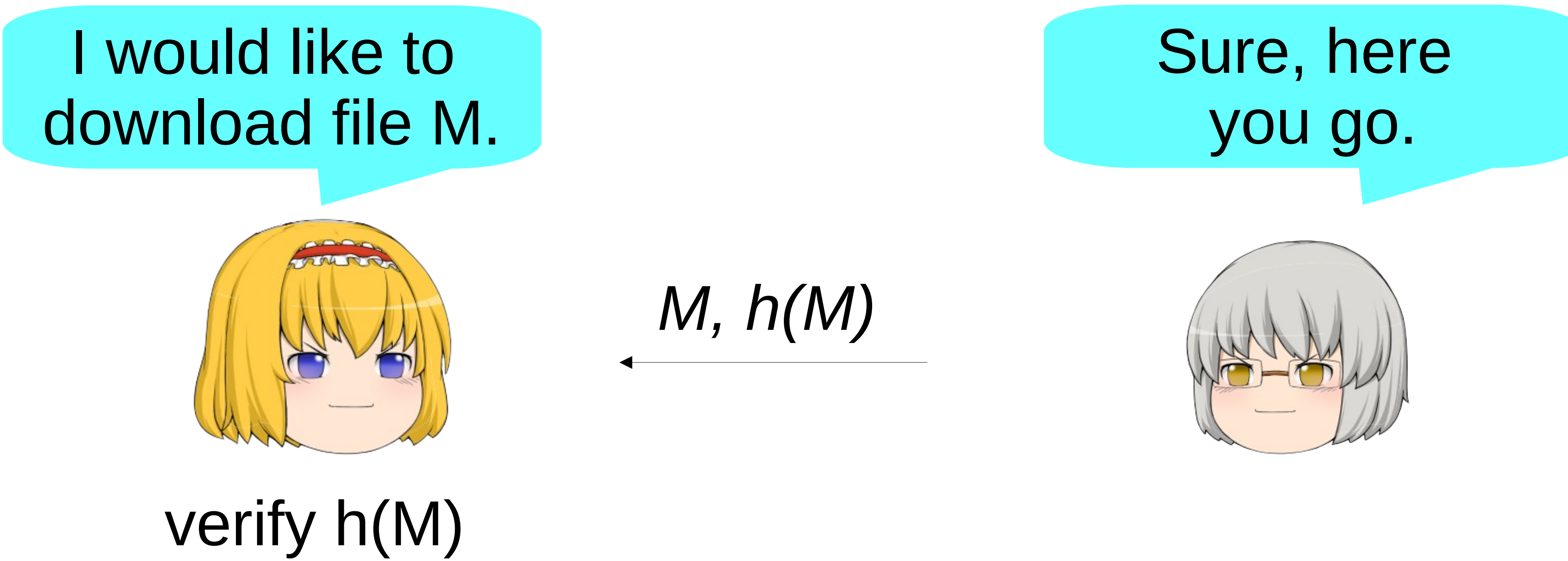
When hashed passwords are stolen, attacker simply has to do a matching exercise to “invert” the hash!

This is exactly like the ciphertext repetition problem

Cryptographic Hash
Password Storage



Cryptographic Hash
Verifying Integrity



Good against unintentional errors, random errors

What about a malicious MITM attacker?

Message Authentication Code

A MAC is attached to messages for authentication:

- The two parties both need to have the secret key (like SKE)
- An attacker cannot “forge” a MAC
- **Authenticates** the message
- Can be built from a hash (this is called HMAC):

$$h(K||M)$$

Message Authentication Code

Alice sends M, h(K||M) to Bob

Verification: Bob, using his key, verifies h(K||M) is correct.

Resistance against MITM: Mallory, who does not have the key, cannot produce the HMAC. Specifically, if Mallory changes M to M’, he cannot also replace h(K||M) with h(K||M’). If he attempts to change any part of the message, Bob’s verification will fail.

Signatures

What if we reversed the roles of  and  ?

~~“Encrypting”~~ would be limited but everyone could ~~“decrypt”~~

Signing verify

 Private signing key: signs the message

 Public verification key: verifies the message

Achieves authentication if you know the correct public verification key

In practice, sign/verify keys are long-lasting while encrypt/decrypt keys are short-lived

Public Key Infrastructure

Hello! I am Alice.
Here is my signature!
Sign *(h(M))*

You can verify it with this public verification key.

How can you trust Alice?

Public Key Infrastructure

Delivering the right public verification key to users

We will examine PKI in three technologies:

- SSH tunneling
- PGP
- SSL/TLS

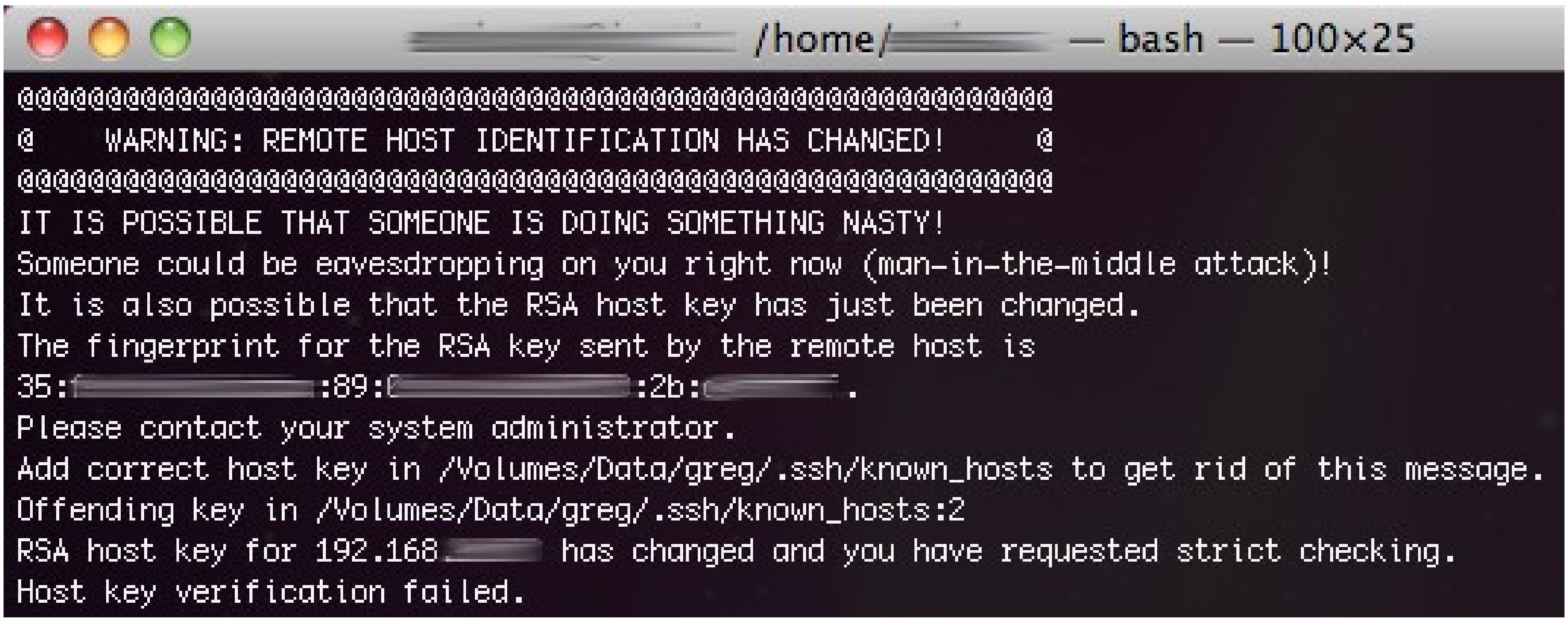
SSH tunneling

Used for connecting to remote machine

TOFU (Trust On First Use):

- When connecting for the first time, the server shows the public key
- You are asked if you trust the public key (yes/no)
- If “yes”, you will not be asked again unless the key changes
- If “no”, you will be disconnected

SSH tunneling



PGP

Used in e-mails

Pretty Good Privacy

- Developed in 1991
- Needs setup
- Used by some professionals, privacy-sensitive circles

PGP

Used in e-mails

Web of Trust:

- Trust is transitive
- Alice can trust Bob directly (like TOFU)
- Alice can trust Carol indirectly – if Alice trusts Bob, and Bob trusts Carol
- Bob signs Carol's key, and Alice verifies Bob's signature

SSL/TLS

Used in HTTP

- Most widely used crypto-technology
- First appeared in Netscape for e-commerce
- Used by default in (increasingly) many websites
- Uses almost all of the tools in this module
- Versions: SSL1, SSL2, SSL3, TLS1.0, TLS1.1, TLS1.2, TLS1.3
- Current trend: removing bad encryption

SSL/TLS

Certificate system:

- By default, browsers will trust a set of **Ceritifcate Authorities** (CA)
- CA can sign any website's public key; the CA's signature is called a certificate
- The website presents its certificate when you connect to it
- Certificates can also be transitive

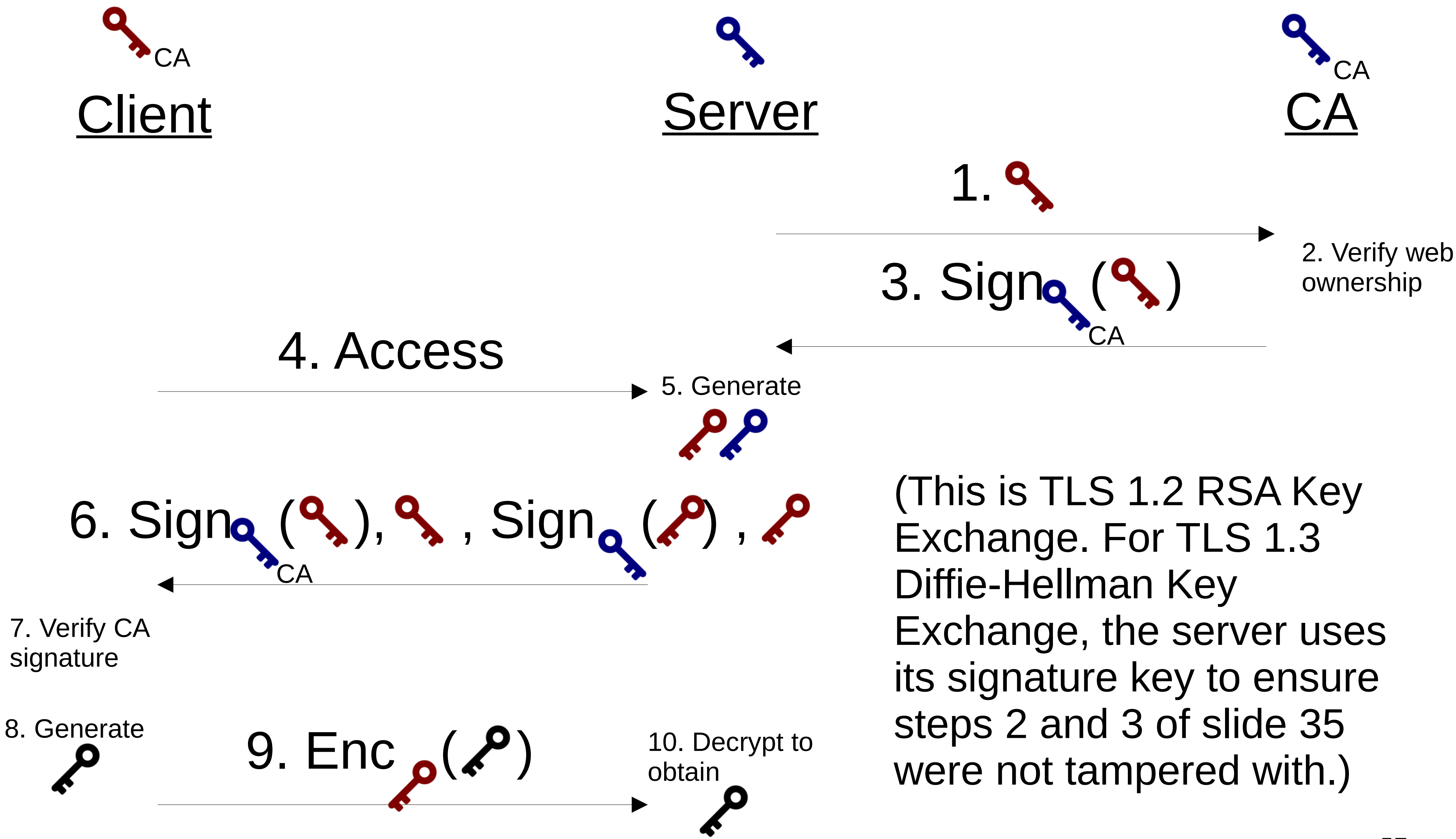
SSL/TLS

A basic connection uses most of this module's tools.

Key:

- CA Root CA’s public verification key
- CA Root CA’s private signature key
- Web server’s public verification key
- Web server’s private signature key
- Web server’s public encryption key
- Web server’s private decryption key
- Secret key negotiated between client and web server

SSL/TLS



(This is TLS 1.2 RSA Key Exchange. For TLS 1.3 Diffie-Hellman Key Exchange, the server uses its signature key to ensure steps 2 and 3 of slide 35 were not tampered with.)

SSL/TLS

1. Server sends its public verification key to the root CA.
2. Root CA checks that person really owns the web server.
3. Root CA signs the web server's public verification key and sends it back (the cert).

(After some time)

4. The client accesses the web server.
5. Server generates an ephemeral PKE key pair.
6. Server sends the cert to client, along with both public keys and a signed version of the public encryption key to avoid tampering.
7. Client checks signature on cert to verify the server's public verification key, then uses that to verify the server's public encryption key.
8. Client generates secret key.
9. Client encrypts secret key with server's public encryption key and sends it to server.
10. Server decrypts to obtain secret key.

From this point onward all communication will use that secret key (most likely 128-bit AES CBC with SHA-256 for HMAC).

Attacks on Cryptosystems

Cryptanalysis

- Find mathematical weaknesses in cryptography
- For example:
  - DES key length is too short
  - RC4 does not have enough initial rounds
  - MD5, SHA-1 are vulnerable to a “collision attack”
    - This is a problem for HMACs

Attacks on Cryptosystems

Root CA compromise:

- DigiNotar, dutch root CA (2011)
  - Issued fake certs for google.com
  - Breach was hidden
  - Web browsers removed DigiNotar as root CA
- Comodo (2011)
  - Issued fake certs for google, yahoo, etc.
  - Certificates were immediately revoked
- Kazakhstan's government issues all certificates (i.e. can read/intercept all HTTPS)

Attacks on Cryptosystems

Yahoo hacks (2014, 2016, 2017)

- Authentication cookies are generated from secret seeds
- Seeds were stolen at some point
- User data stolen by criminals, sold several times

The stolen user account information may have included names, email addresses, ..., hashed passwords ***(using MD5)****...*

Recap

SKE	is efficient and hard to break cryptographically
	but it needs a shared key
PKE	can be used to share the SKE key
	but text and public key are not authenticated
MAC	can authenticate text
	but it needs a shared key
PKI	can authenticate public key
TOFU *Web of Trust* *PKI*	*but they each have their own problems*