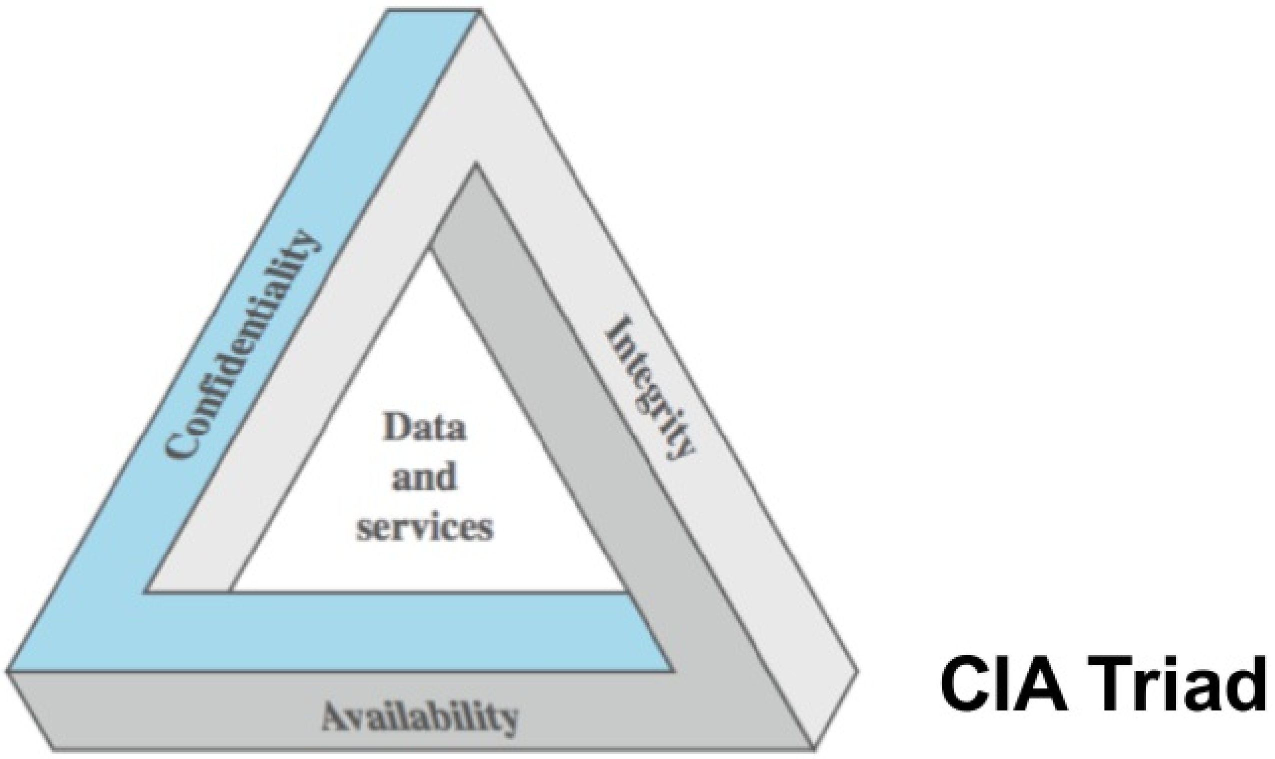
IntroSec - Summary

Why is security so difficult?

- Functionality
  - if user does *expected input*
  - then system does expected action
- Security
  - if user/outsider does unexpected things
  - system does not do any really bad action
- Why difficult
  - what are all possible unexpected things? possibly unbounded
  - how do we know that all of them are protected?
  - At what level of system abstraction?
- Security properties are undecidable!

Computer security

Protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity and confidentiality of information system resources (includes hardware, software, firmware, information, data, telecommunication)



Key Security Concepts - Terminology

- **Confidentiality**
  - **Data Confidentiality**: assures private/confidential information is not made available/disclosed to unauthorized

    individuals
  - **privacy**: assures individuals control or influence what information related to them may be collected/stored and by whom that information may be disclosed

- **Integrity**

- **Data integrity**: assures information and programs are changed only in specified/authorized manner

- **System integrity**: assures system performs its intended function in an unimpaired manner

- **Availability**: assures systems work promptly and service is not denied to authorized users

- **Authenticity**: property of being genuine and being able to be verified/trusted; confidence in the validity of a transmission message / message originator

- **Accountability**: security goal that generates requirement for actions of an entity to be traced uniquely to that entity

- **RFC 2828, Internet Security**

  - **Security policy:** set of rules/practices to be followed in order protect sensitive/critical resources
  - **System resources(asset):** smth. worthy to be protected
  - **Threat:** potential violation of security
  - **Vulnerability:** flaw/weakness in system design

Scope of Computer Security

- Hardware
  - availability(damage, theft)

- Software
  - availability(deletion, damage)
  - integrity(modification through, e.g. malware)

- Data
  - availability(destruction of data files)
  - integrity(modification of logs/passwords)
  - confidentiality(unauthorized reading, deliberately released statistics)

Security Functional Requirements

- **Technical measures:**
  access control; identification/authentication; system and communication protection; system/information integrity; verification
- **Management controls and procedures:**
  awareness/training; audit/accountability; certification; accreditation, security assessments; contingency planning; maintenance; physical/environmental protection; personnel security; risk assessment systems; service acquisition
- **Overlapping technical and management:**
  configuration management; incident response; media protection

Vulnerabilities and Attacks

- Vulnerabilities make system resources
  - corrupted(loss of integrity)
  - leaky(loss of confidentiality)
  - unavailable(loss of availability)

- Attacks are threats carried out and may be
    - passive: learn/make use of system information but not affect system resources
    - active: alter system resources or affect their operation
    - insider: initiated by entity inside the security perimeter
    - outsider: initialized by unauthorized/ illegitimate user of the systems

Security Principles

- Simplicity
- Open Design
- Compartmentalization
- Minimum Exposure
- Least Privilege
- Minimum Trust, Maximum Trustworthiness
- Fail Secure
- Complete Mediation
- No Single Point of Failure
- Traceability
- Randomness
- Usable security

Simplicity

Keep it simple, stupid (KISS principle)

- Applies to any engineering/implementation task:
- The simpler the solution
    - the easier to understand, analyze, review
    - less likely to contain flaws

Open design

The security of a system should not depend on the secrecy of its protection mechanism

- Avoid *security by obscurity*
- Security should depend on possession of secrets only (passwords, keys, ...)
    - not possible to maintain secrecy of a system that should be distributed

- Today's de-facto crypto mechanisms all developed with open design

Compartmentalization

Organize resources into isolated groups of similar needs

- Groups (or compartments) isolated from each other with limited communication between compartments over controlled channel
- Facilitates simplification of design; attacks/errors contained to affected compartment; security-sensitive functionality can be in dedicated hardened compartment
- **Compartmentalization** at different levels:
    - user space vs. kernel space
    - memory space (between processes; data vs code)
    - modularization of software

    - virtual machines
    - network zones

- Problem: not always possible to completely isolate resources/functionality
    - tightly control channel between compartments and their interfaces

Minimum Exposure

Minimize the attack surface a system presents to the adversary

- reduce external interfaces to a minimum
- limit amount of information given away that can help an adversary
- minimize window of opportunity for adversary to attack

Least Privilege

Any component (and user) of a system should operate using the least set of privileges necessary to complete its job

- **privilege**
    - ability to access/modify a resource
    - *privileged process:* process that has access to some resource not generally available
    - more secure systems have many types of privilege

- malicious or compromised process cannot misuse privileges that it does not have
    - compartmentalization helps realizing least privilege
    - delegation of tasks to sufficiently processes, requires careful design of programs to avoid *confused deputy attacks*(attacker fooling a privileged process to misuse authority)

- implementation often difficult: requires detailed understanding of systems and all possible operations/dependencies

Minimum Trust and Maximum Trustworthiness

Minimize trust, maximize trustworthiness

- **Trusted system**: user assumes that system will behave as expected, but system may misbehave(by acting maliciously)
    - avoid trust when possible

- **Trustworthy system**: system always acts as expected by user
- **minimizing trust** = minimizing expectations about system
- **maximizing trustworthiness** = turning assumptions into validated properties
- Problem: transitive trust (*chain of trust*)
    - A $\rightarrow$ B, B $\rightarrow$ C, thus A &rightarrow C (possibly unknown to A)
    - complicates reasoning about security of an entity

Secure, Fail-Safe (Fail-Secure) Defaults

The system should start in a secure state and return to a secure state in the event of a failure

- system designed to prevent unsafe/insecure consequences of system's failure
    - requires start in secure state

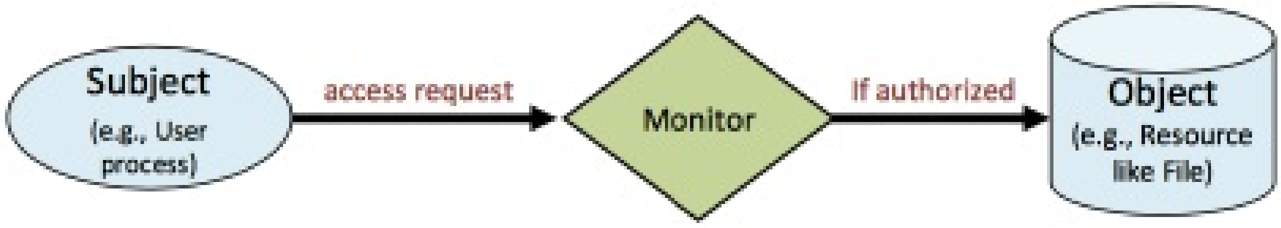
- important in access control
    - identify conditions under which access is granted
    - default case: if conditions not fulfilled, deny access

- fail securely: conditions wrongly identified $\rightarrow$ access denied (wrongly)

Complete Mediation

Access to any object must be monitored and controlled

- requires that access control mechanism encompasses all security-relevant objects and is operational in any system state



- access control must not be circumventable
- to mitigate attacks at layers below access control, protect data additionally in transit/storage
- requires authentication to securely identify subjects/objects of any access

No Single Point of Failure

Build redundant security mechanism whenever feasible

- also known as **defense in depth**: do not rely on single security mechanism, if one fails, another should prevent malice
- how many layers?
  depends on cost-benefit analysis: performance requirements, usability aspects, administrative overhead, etc.
- common technique: **Separation of duties**:
  more than one entity/mechanism required to complete task

Traceability

Log security-relevant system events

- **trace**: sign or evidence of past events
  - **traceability**: System retains traces of activities
- important for accountability:
  link actions to subject that can be held responsible
- usually implemented through **logging**: logs must be protected to prevent adversary from removing traces
- can contradict privacy requirements
  - possible solution: use of pseudonyms, store true identities separately, ...
  - however careful: pseudonyms often linkable, thereby breaking privacy

Randomness

Maximize the entropy of secrets

- **Entropy** in information theory: *degree in randomness*
- high entropy of secrets required to prevent guessing/brute-forcing

Usable Security

Design usable security mechanisms

- security mechanism should be easy to use
  - the harder to use, the more likely it is that users(/developers/admins) will circumvent it or apply it incorrectly

Authentication

- Identification = whom you claim to be
- Authentication = how you prove your ID
  - credentials: evidence used to prove ID
  - usually prerequisite for authorization to use system resources
- Forms:
  - something you know (password/pin)
  - something you have (smart card, hardware token, ...)
  - something you are (fingerprint, face ID)
  - combination of those (two factor authentication)

Password Authentication

- most common method
- issues:
  - how is password stored?
  - how does system check PW?
  - how easy is it to guess a PW?

Hash functions

users password → hash fuction → password file
hash-function(arbitrary length input) → fixed length output
Cryptographic hash functions have following properties:

- One-way-functions
  easy to compute output from input, infeasible to compute input from output
- collision-resistance
  infeasible to find two different inputs that map to same output
- hash function h
  - user password stored as h(password)
  - when user enters passwords
    - system computes h(password)
    - compares entry in password file
  - n plaintext password stored

Advantages of Salting

structure of password entry (UNIX): `username:$id$salt$h(salt|password)$`

- $: separator between entry fields
- id: identifies algorithm h used to compute hash
- salt: random for every user

without salt:

- attacker can pre-compute hashes of all common passwords once
- same hash function on all UNIX machines: identical passwords hash to identical values
- one table of hash values works for all password files
- *"break once, break everywhere"*

with salt:

- attacker must compute hashes for all common password for each possible salt value
- dictionary attack for known password file
  - for each salt found in file, try all common strings
  - important: salt like password hash are publicly readable

Biometrics

- Use person's physical characteristics
- advantages:
  - cannot be disclosed, lost, forgotten
- disadvantages:
  - cost, installation, maintenance
  - reliability of comparison algorithms
    - false positive: allow access to unauthorized person
    - false negative: disallow authorized person
  - privacy?
  - if forged, how do you revoke it?

Hardware Authentication Tokens

- generates **one-time passwords** or **challenge-response authentication** using secure hardware key
- devices communicate via USB, NFC, etc.
- advantages
  - no need to remember password
  - password reuse is no issue
- disadvantages
  - what if device is lost?
  - you have to bring device always with you
  - expensive

Two-Factor Authentication(2FA)

- combine 2 types of authentication
- second factor to work around limitations of fist
- important security requirement:
  both channels independent from each other;
  so compromise of one channels does not comp. the other

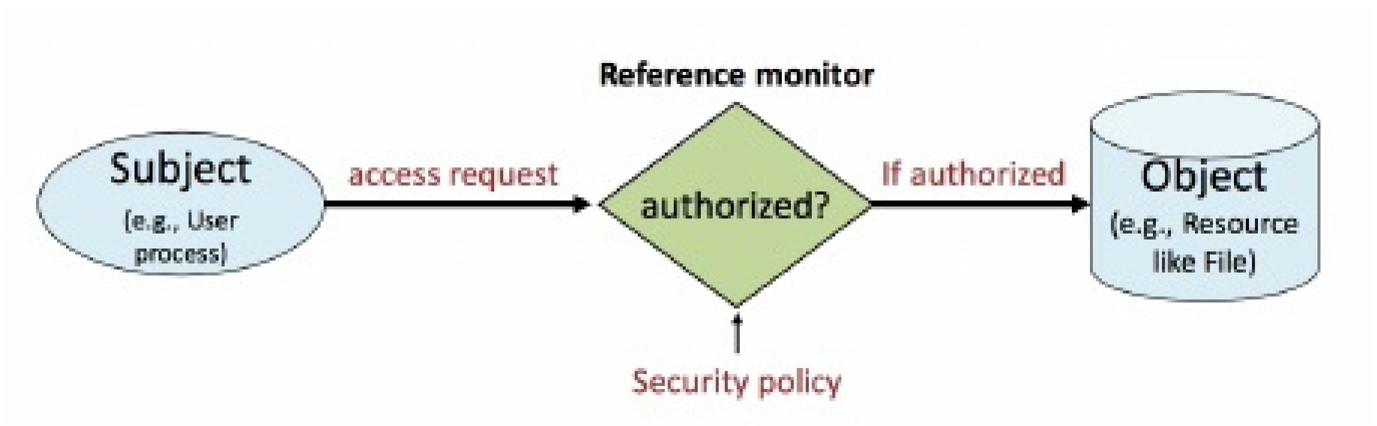
Access Control

Access Enforcement Mechanism

- authorizes *requests by subject* to perform *operations on objects*
- 2 important components:
    - **reference monitor**: access enforcement mechanism
    - **security policy**: rules to describe operations a subject can perform on objects, defines how rules can be modified

Reference monitor

- **assumption**: system knows who subject is
- access requests pass through gatekeeper ("reference monitor")
- subject needs to fulfill well-defined properties



- **properties**:
    - complete mediation: system must not allow monitor to be bypassed
    - tamperproof: monitor must be protected from compromise
    - verifiable: monitor should be small enough to be analyzed

How to store access rights

- **Access Control List(ACL)**
    - object-centered: associate each object with list
    - reference monitor checks subject against list of accessed object
    - relies on authentication: need to know user

- **capabilities**
    - subject-oriented: capacity is unforgettable ticket(token) that defines privilege of holder
        - usual implementation: random bit sequence, of managed by OS
        - can be passed from one process to another

    - reference monitor checks token
        - does not need to know identity of user/Process

Roles (Groups)

- **Role** = set of users
    - administrator, powerUser, users, guest
    - assign permission to roles

- **Role hierarchy**
    - partial order of roles
    - each role gets permission of roles below
    - list only new permissions given to each role

Access Control Types: DAC vs. MAC

- Discretionary Access Control (DAC):
  - subjects can freely delegate/revoke/modify access rights to objects for which they have access rights

- Mandatory Access Control (MAC):
  - security policy set and modified centrally by trusted admin
  - users/untrusted processes cannot override policy
  - subjects/objects usually labeled with security attributes
  - access rules describe allowed operations between labels
  - transition rules describe how labels of subjects/objects can change

Multi-Level Security (MLS) Concepts

- Military security policy
  - classification involves sensitivity levels, compartments
  - do not let classified information leak to unclassified files

- group individuals and resources
  - form of hierarchy to organize policy

- Bell-LaPadula Model
  - primarily for military/government MLS
  - control information confidentially: "no read up, no write down"
    - subject cannot read an object of higher security level
    - subject cannot write to object of lower security level: no accidental downgrade of information

- Conditional Policies
  - temporal policies
  - context-aware policies

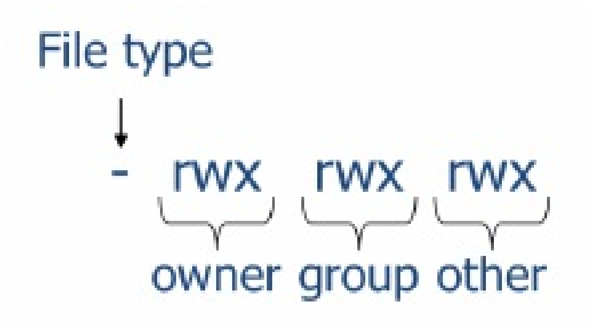
- Biba Model
  - Dual of Bell-LaPadula
  - preservation of data integrity:
    - subject cannot read object of lower level
    - subject cannot write object of higher level

- Separation of Duty
- Chinese Wall Policy

Unix File System

- Each file has owner and group
- permissions set by owner
  - access rights: read(r), write(w), execute(e)
  - subjects: owner, group, other
  - representation: symbolic, bit field in octal notation

- only owner and root can change permissions



- owner can have fewer privileges than other
- prioritized resolution of differences
    - if user = owner : owner permission applies
      else if user in group : group permission applies
      else : other permission

summary

- good things:
    - some protection from most users
    - flexible enough to make thinks possible
- main bad thing:
    - too tempting to use root privileges
    - no wat to assume some root privileges without all root privileges

Memory attacks

control-flow hijacking attacks

- Attacker's goal:
    - take over target machine
    - execute arbitrary code on target by hijacking application control flow
- pattern is always similar
    1. find bug in program
    2. create code to exploit bug and control program
    3. feed vulnerable program with exploit
    4. hijacked

computer programs: assumptions

- we write code in languages that offer several layers of abstraction over machine code
- naturally, our execution model assumes:
    - basic statements are atomic
    - only one of the branches of if statement can be taken
    - functions start at beginning
    - execute from beginning to end
    - when done, they return to call site
    - only code in program can be executed
    - set of executable instructions is limited to those output during compilation of the program
- but actually at level of machine code:
    - each basic statement compiled down to main instructions
    - no restriction on target of a jump

  - can start executing in middle of functions
  - fragment of function may be executed
  - returns can go to any program instruction
  - dead code can be executed

Buffer Overflows

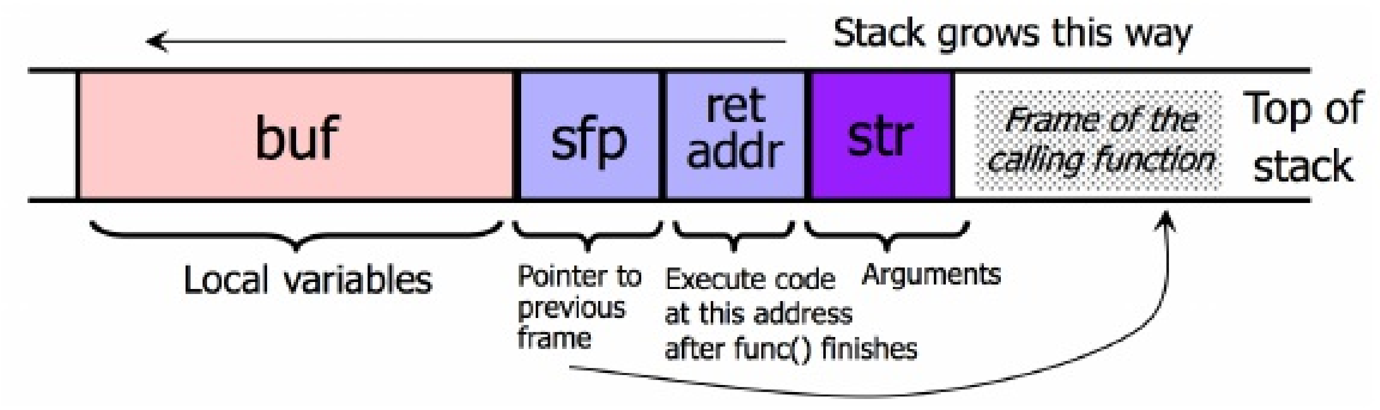
common assumption: target buffer is large enough for source data

example

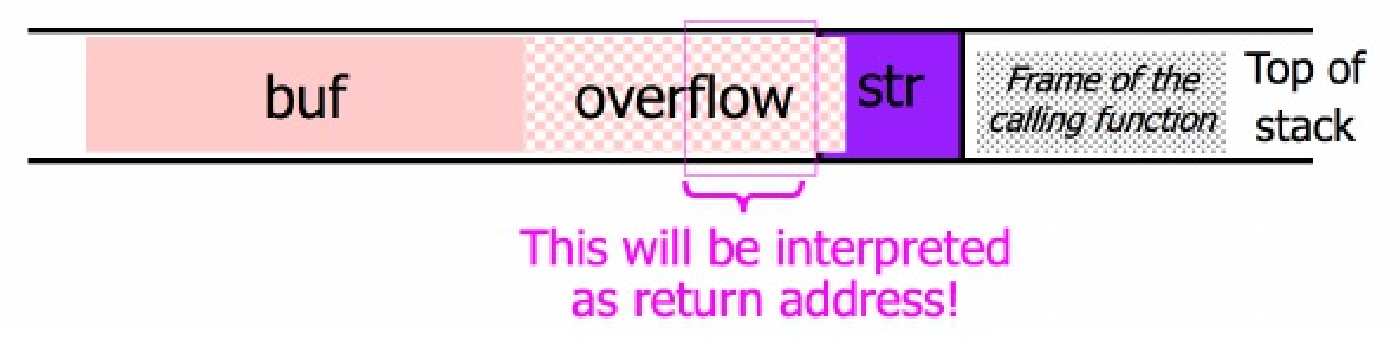
- given function:

```
void func(char *str) {
  char buf[126]; //allocate local buffer
  strcpy(buf, str); //copy argument to local buffer
}
```

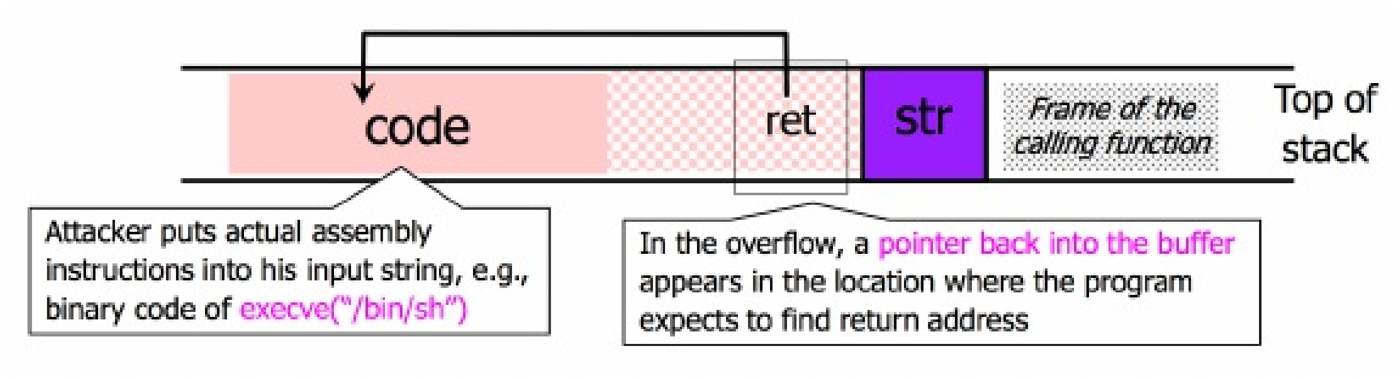
- when function is invoked, a new frame is pushed onto the stack


Stack grows this way
buf
sfp
ret addr
str
Frame of the calling function
Top of stack
Local variables
Pointer to previous frame
Execute code at this address after func() finishes
Arguments

- memory pointed to by `str` is copied onto stack, but it is not checked whether the string at `*str` contains fewer than 126 characters
- if a string longer than 125 bytes is copied into buffer, it will overwrite adjacent stack locations


buf
overflow
str
Frame of the calling function
Top of stack
This will be interpreted as return address!

- suppose buffer contains attacker-created string


code
ret
str
Frame of the calling function
Top of stack
Attacker puts actual assembly instructions into his input string, e.g., binary code of execve("/bin/sh")
In the overflow, a pointer back into the buffer appears in the location where the program expects to find return address

- when function exits, code in buffer will be executed, giving attacker a shell

Memory Defences & More Complex Attacks

Memory Exploits

- Buffer is a data storage area inside computer memory
  - intended to hold pre-defined amount of data
  - simplest exploit: supply executable code as "data", trick victim's machine into executing it
- overwriting adjacent memory locations can
  - modify other variables (corruption of data)
  - modify control flow data such as return addresses and pointers to previous stack frames
- in the worst case, attacker will execute arbitrary code with the privileges of the attacked process

Memory Exploit Causes

- Assembly does not provide any notion of type
- languages such as java, Python are memory-safe
  - strong notion of types
  - overflows are not possible
- c and c++ are memory unsafe
- buffer overflows can be caused by
  - reading data from stdin or the network
  - copying/merging data
  - bugs in boundary check
  - appending strings
  - ...

Stack Canaries

- Embed **canaries**(aka stack cookies) in stack frames and verify their integrity prior to function return
  - any overflow of local variables will damage canary
- make it hard to exploit a buffer overflow

Three kinds of canaries

- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

    - Disadvantage: canary is known → attacker can overwrite it with the known value

- Random canary: random string chosen at program start, stored in global variable padded by unmapped pages
    - attacker can't guess value of canary and any try to exploit bugs to read off RAM will cause segmentation fault
    - disadvantage: the attacker can still learn the canary if he knows where to read it from

- Random XOR canary: random string chosen at program start, XOR scrambled using all or part of control data to protect
    - can be used to detect attacks in which attacker is able to modify return address without overwriting canary
    - disadvantage: same Vulnerability as random canaries, but attacker also needs to know control data and scrambling algorithm

Stack Canaries: Cons and Limitations

- requires code recompilation
- checking canary integrity prior to every function return causes performance penalty
- StackGuard can be defeated
- Do not prevent heap-based buffer overflows
- only protect against contiguous buffer overflows
- no protection if attack happens before function returns
    - canary wont detect if exploit overwrites

- canary alone offers no protection for local pointers

Buffer Overflow: Causes and Cures

- "classic" memory exploit involves *code injection*
- idea: prevent execution of untrusted code
    - make stack and other data areas non-executable
    - digitally sign all code
    - ensure that all control transfers are into a trusted, approved code image

W⊕X/DEP

- mark all writable memory locations as non-executable
    - W⊕X (Write XOR execute)

- **issues:**
    - some applications require executable stack
    - JVM makes all its memory RWX (readable, writable, executable)
    - attack can start by "returning" into memory mapping routine, make page containing attack code writable

- can still corrupt stack
    - or function pointers or critical data on heap

- as long as "saved EIP" (Extended Instruction Pointer) points into existing code, W⊕X protection will not block control transfer

Return-to-Libc Exploits

- Overwrite saved EIP with address of any library routine, arrange memory to look like arguments
- does not look like huge threat at first:
    - attacker cannot execute arbitrary code
    - ... especially if system() is not available

- overwritten saved EIP need not point to beginning of library routine
- any existing instruction in code image is fine
  - system will execute the sequence starting from this instruction

EBP and ESP Pointers

- EBP = extended base pointer
- ESP = extended stack pointer
- local variables can be accessed by referencing EBP

Address Space Layout Randomization(ASLR)

Problem: Lack of Diversity

- classic memory exploits need to know address to hijack control
  - address of attack code in buffer
  - address of standard kernel library routine

- same address is used on many machines
- idea: introduce artificial diversity
  - make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

ASLR

- randomly choose base address of stack, heap, code segment, location of Global Offset Table
- randomly pad stack frames an malloc'ed areas
- implemented in today's OSs
- effectiveness depends on amount of randomness
- crucial to keep pointer information secret: if attacker can read, we are screwed

Short summary on hardening techniques

- *Stack canaries* protect only against contiguous overwrites of the stack
- *DEP* protects against code injection but not against code reuse
- *ASLR* does not protect against information leakage and it can anyway be bypassed

Browser Security

Background

Web applications

1. user requests webpage with dynamic content
2. web application queries database
3. DB data used by application to generate page content
4. received data is rendered by client's browser

Browser: basic execution model

- each browser window or frame:
  - loads content
  - renders pages

    - process HTML/scripts to display page
    - may involve images, subframes, etc.

- events
  - user actions: onCllick, OnMouseover
  - rendering: OnLoad, OnUnload
  - timing: setTimeout(), ...

JavaScript in web pages

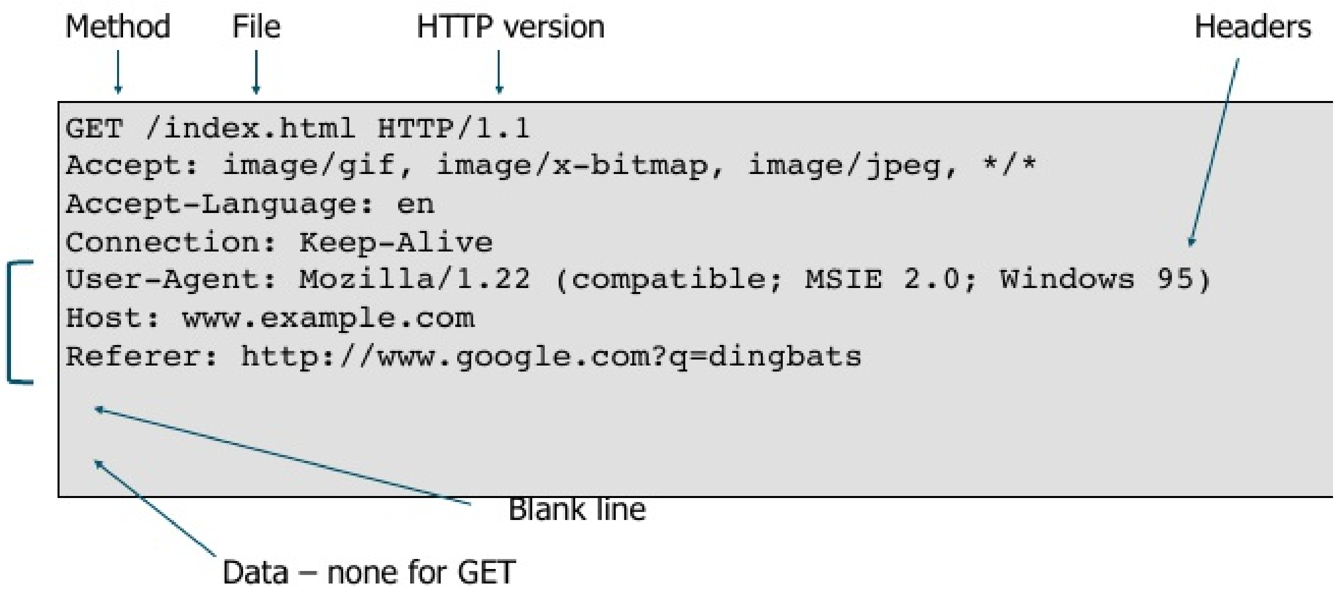
- embedded in HTML page as <scipt> element
  - javaScript written directly insice <sript> element
  - linked file as src attribute of <script> element
- event handler attribute
- pseudo-URL references by a link

DOM and BOM

- JavaScript can interact with HTML page and browser through DOM and BOM
- Document Object Model (DOM)
  - object-oriented representation of hierarchical HTML structure
  - properties
  - methods (change content of page)
- Browser Object Model (BOM)
  - window, Frames[], history, location, navigator(type/version of browser)

HTTP protocol

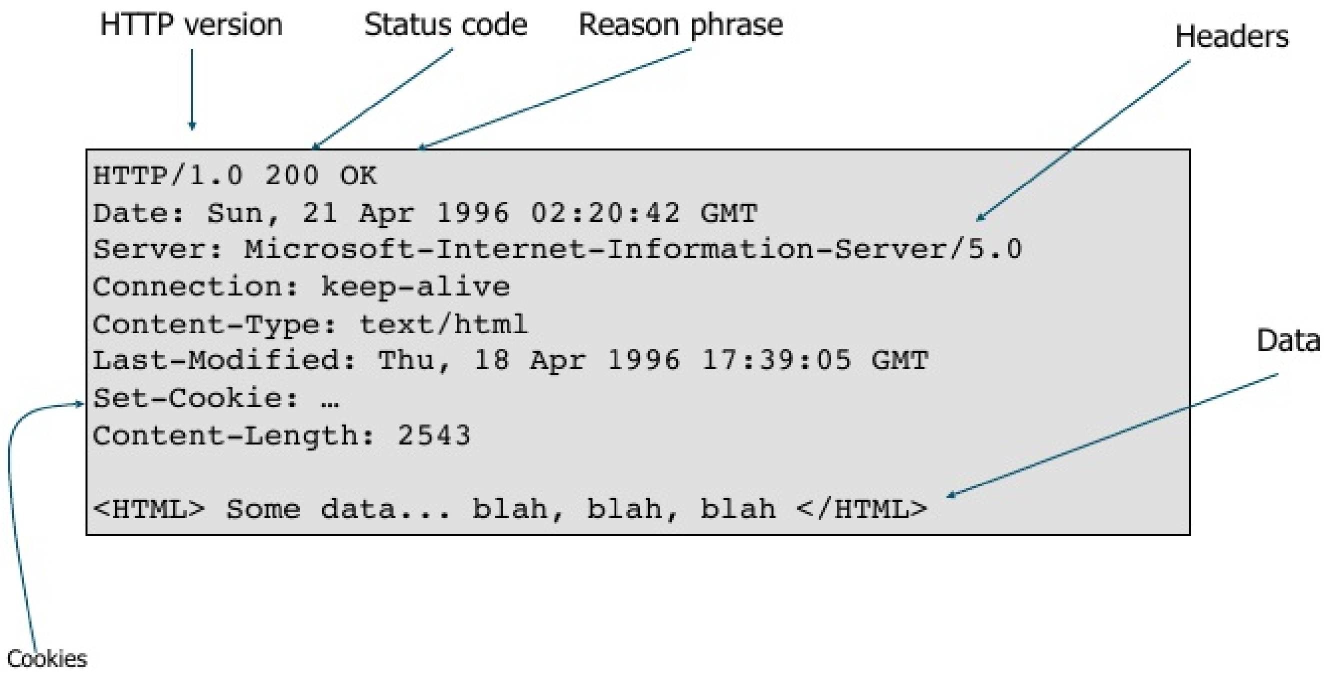
- Hypertext Transfer Protocol
  - widely used
  - simple
  - Port 80
  - stateless
  - unencrypted
- Hypertext Transfer Protocol Secure (HTTPS)
  - port 443
  - encrypted by SSL/TLS
- **Uniform Resource Locator(URL)**
  - global identifiers of network-retrievable documents
- **HTTP Request**



```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

- GET: no side effect, used to retrieve data
- POST: possible side effect, insert/update remote resources

- **HTTP Response**



```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: …
Content-Length: 2543

<HTML> Some data... blah, blah, blah </HTML>
```

Web application security problem

- delusive simplicity for creating web apps
- lack of security awareness
- time- & resource limits during development
- rapid increase in code complexity

Web attacker

- controls a malicious website (attacker.com)
  - can even obtain SSL/TLS certificate for site
- user visits attacker.com
  - phishing email, enticing content, search results, ...
  - attacker's facebook app

- attacker has no other access to user machine!
- variation: "iframe attacker"
  - iframe with malicious content included in otherwise honest webpage (advertising, mashups)

Network and malware attacker

- Network attacker
  - passive: wireless eavesdropper
  - active: evil Wi-Fi router, DNS poisoning
- Malware attacker
  - malicious code executes directly on victim's computer
  - to infect victim's computer, can exploit software bugs or convince user to install malicious content

Common web attacks

- client-side attacks:
  - cross-site scripting (XSS)
  - cross-site request forgery (CSRF)
- server-side:
  - SQL injection
  - path traversal
  - remote code injection

Goals of web security

- safely browse the web
- malicious website cannot steal information from or modify legitimate sites or otherwise harm user
- even if visited concurrently with a legitimate site - in separate window, tab, iframe

Same Origin Policy

browser sandbox
goal: safely execute JavaScript code provided by remote website by enforcing isolation from resources provided by other websites
same origin policy
can only read properties of documents and windows from the same protocol, domain, and port

- SOP for DOM: Origin A can access origin B's DOM if A and B have same (protocol, domain, port)
- SOP for cookies: Generally based on ([protocol(optional)], domain, path)

often misunderstood

- often simply stated as "same origin policy"
- full policy of current browsers is complex
  - evolved via "penetrate-and-patch"
  - different features evolved in different policies
- common scripting and cookie policies
  - script access to DOM considers protocol, domain, port
  - cookie reading considers protocol, domain, path
  - cookie writing considers domain

Cookies

file created by website to store information in the browser
HTTP is a stateless protocol, cookies add state

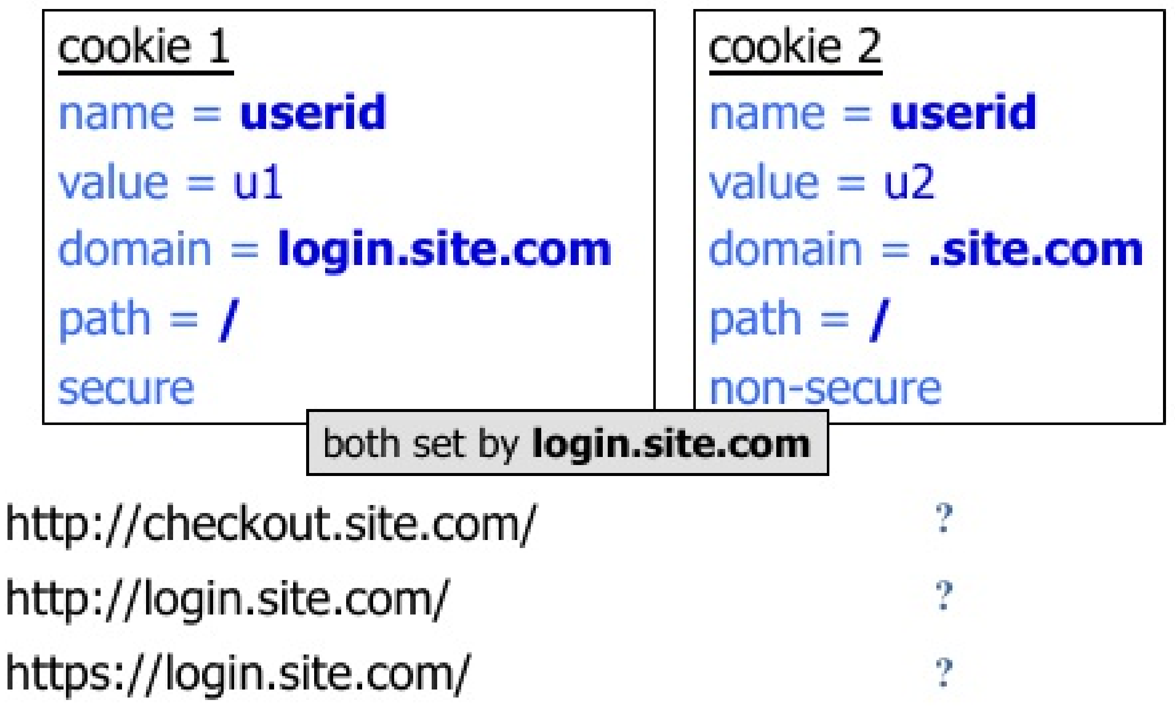
- authentication: cookie proves that client previously authenticated correctly
- personalization: helps website recognize user from previous visit
- tracking: follow user from site to site; learn browsing behavior, preferences, etc.

SOP for writing cookies

- domain: any domain suffix of URL-hostname, except top-level domain
- path: anything

SOP for reading cookies
browser sends all cookies in URL scope:

- cookie-domain is domain-suffix of URL-domain
- cookie path is prefix of URL-path
- protocol=HTTPS if cookie is "secure"



SOP for JavaScript in Browser

- same domain scoping rules as for sending cookies
- `document.cookie` returns string with all cookies available for document
    - often used to customize page
- javaScript can set/delete cookies via DOM
    - `document.cookie = "name=value; expires=...;"`

path separation is not secure

- cookie SOP: path separation
    - when browser visits *x.com/A*
    - it does not send cookie of *x.com/B*
    - done for efficiency, not security

- DOP SOP: no path separation
  - script from *x.com/A* can read DOM of *x.com/B**

HttpOnly Cookies
cookie sent over HTTP(s), but not accessible to scripts

- cannot be read via document.cookie
- blocks access vom XMLHttpRequest headers
- helps prevent cookie theft via XSS
- does not stop most other risks of XSS bugs

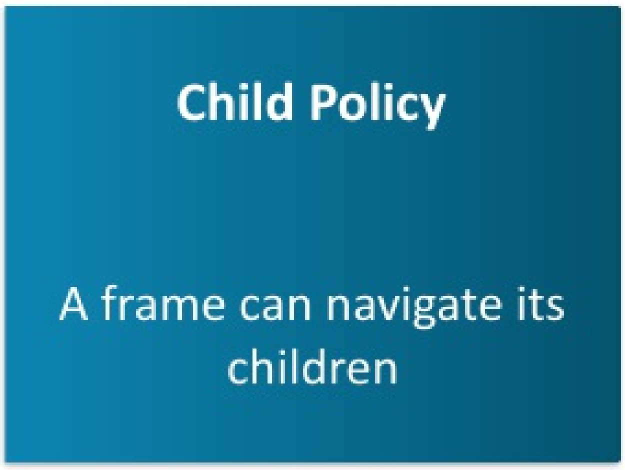
Frames

- window my contain frames from different sources
  - frame: rigid division as part of frameset
  - iframe: floating inline frame
- modularity
  - brings together content from multiple sources
  - client-side aggregation
  - delegate screen area to content from another source
- browser provides isolation based on frames
  - different frames can represent different principals
  - can't script each other
  - frame can draw only on its own rectangle
- robustness: parent ma work even if frame is broken
- browser security policies:
  - each frame of a page has an origin
  - frame can access objects from own origin: read/write DOM, cookies, localStorage
  - frame cannot access objects associated with other origins

cross-frame scripting

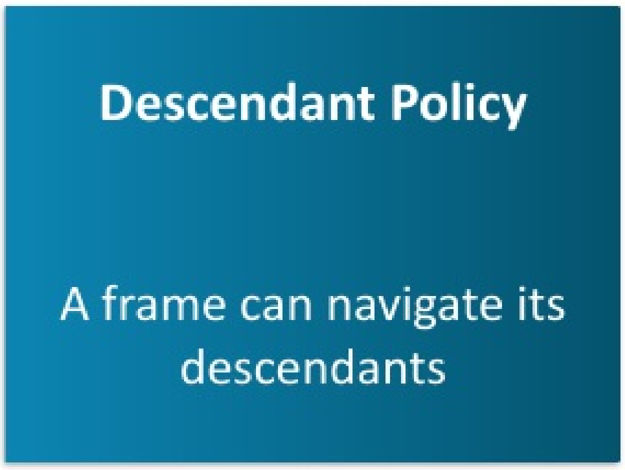
- frame A can execute script that manipulates arbitrary DOM elements of frame B only if origin(A)=origin(B)
- some browsers used to allow any frame to navigate any other frame
  - navigate = change where content in frame is loaded from
  - navigation does not involve reading frame's old content

frame navigation / policy behavior



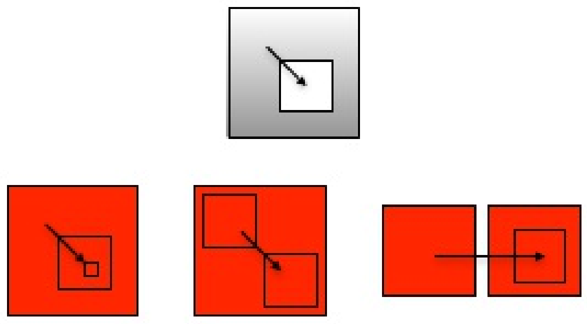
Child Policy

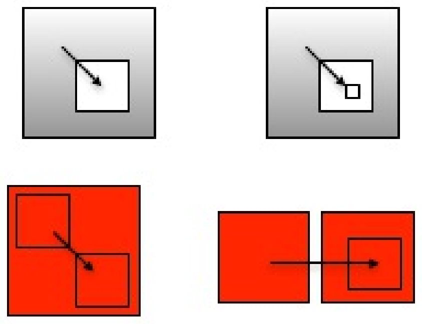
A frame can navigate its children

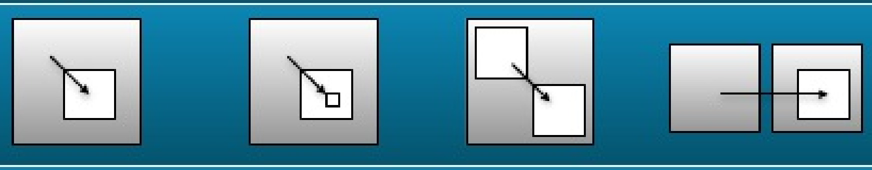
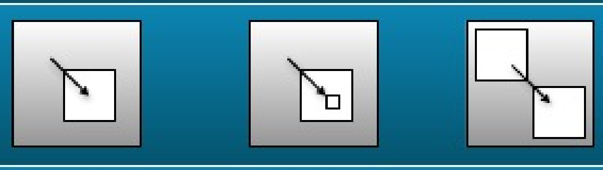
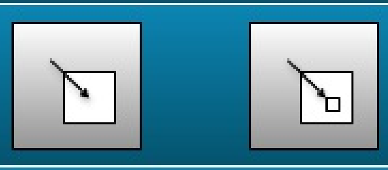
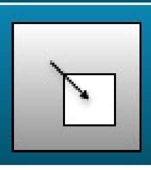


Descendant Policy

A frame can navigate its descendants





Policy	Behavior
Permissive	
Window	
Descendant	
Child	

all popular browsers use descendant policy

CORS

- cross-origin resource sharing (CORS)
  - access-control-allow-origin: <list of domains>
  - typical usage: access-control-allow-origin: *

Client-side messaging via postMessage, message eavesdropping

messages are sent to frames, not origings **message eavesdropping**

1.

- assume descendant frame navigation policy
- attacker embeds integrator in own web page
- integrator call postMessage on gadged → attacker replaces is with own gadged
- attakcer gets secret

2.

- source is reference to frame which sent message
- attacker replaces child frame with own, gets secret message

authors should check origin attribute to ensure that messages are only accepted from domains that they expect to receive messages from

Library (code) import

- SOP does not apply to directly included scripts (not enclosed in iframe)
- script has privileges of imported page, not source server
- can script other pages in this origin, load more scripts

SOP does not control sending

- SOP controls access to DOM
- active content(scripts) can send anywhere
  - no user involvement required
  - can only read response from same origin

Web Security

top web vulnerabilities

- SQL injection (server-side)
  - malicious data sent to website is interpreted as code in query to back-end database

- CSRF(XSRF) - cross-site request forgery (client-side)
  - bad website forces user's browser to send request to good website

- XSS - cross-site scripting (client-side)
  - malicious code injected into trusted context

SQL Injection

- SQL - Structured Query Language
- widely used database query language
- syntax independent of vendor
- basic syntax:
  - fetch set of records: `SELECT * FROM <table_name> WHERE CONDITIONS`
  - insert: `INSERT INTO <table_name> VALUES...`
  - delete table: `DROP TABLE <table_name>`

SQL Injection

- input validation vulnerability
- untrusted user input in SQL query sent to backend database without sanitizing data
- data can be misinterpreted as command

- can alter intended effect of command or query
- example:

```
$user = "admin";
$pass = "' OR '1' = '1";

$sql = "SELECT * FROM  UserTable
                 WHERE User = 'admin' AND
                      Passwd = '' OR '1'='1';
```

'1' = '1' evaluates to TRUE → all user data is leaked

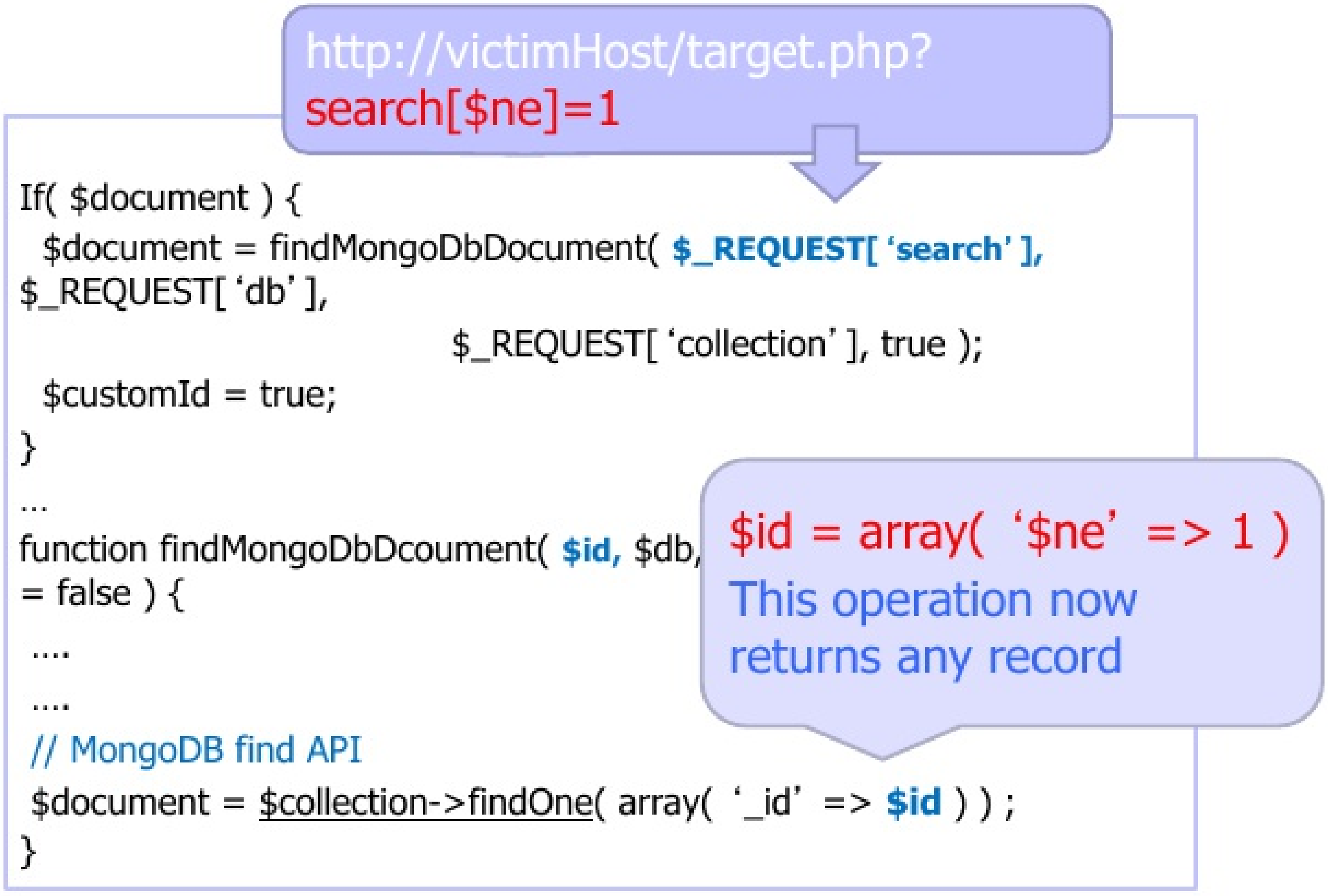
second-order SQL injection

- data stored in database can be later used to conduct SQL injection
- example, user manages to set username to *admin'*--

```
UPDATE USER SET passwd='cracked' WHERE uname='admin'--'
```

    - vulner could occur if input validation/escaping applied inconsistently
        - some web applications only validate inputs coming from server, but not from back-end DB
- solution: treat all parameters as dangerous

NoSQL injection



http://victimHost/target.php?
search[$ne]=1

```
If( $document ) {
  $document = findMongoDbDocument( $_REQUEST[ 'search' ],
$_REQUEST[ 'db' ],
                              $_REQUEST[ 'collection' ], true );
  $customId = true;
}
...
function findMongoDbDcoument( $id, $db,
= false ) {
 ....
 ....
 // MongoDB find API
 $document = $collection->findOne( array( '_id' => $id ) ) ;
}
```

$id = array('$ne' => 1)
This operation now returns any record

wrap up

- problem:
  - insufficient validation of untrusted data
  - never trust user input!
  - validate input data according to specified requirements
- consequences
  - information leakage
  - data manipulation
  - sabotage
- prepared statements
  - in most injection attacks, data interpreted as code
  - bind variables: placeholders guaranteed to be data
  - prepared statements allow creation of static queries with bind variales

Cross Site Request Forgery

- user logs into bank.com, forgets to sign off
  - session cookie remains in browser state
- user then visits malicious website containing

  ```
  <form name=BillPayForm
  action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy>...
  <script> document.BillPayForm.submin();</script>
  ```

- browser sends cookie, payment request fulfilled
- problem: cookie authentication not sufficient when side effects can happen

CSRF Defenses - Secret validation tokens

- synchronizer token pattern (forms)
  - secret, random number embedded by web application in all HTML forms, verified on server side
  - hard to keep in secret against malicious scripts reading webpage
- cookie-to-header token (javaScript)
  - on login, set cookie with random number
  - javaScript reads value, copies it into custom HTTP header
  - server checks token

CSRF Defenses - Referer validation

- **lenient** referer checking - header is optional
- **strict** referer checking - header is required
- why not always strict? referer header suppressed because
  - stripped by organization's network filter
  - stripped by local machine
  - stripped by browser for https $\rightarrow$ http trasitions
  - user preference in browser
  - buggy browser

- web application can't afford to block these users
- referer rarely suppressed over https

CSRF Defenses - Custom header

- custom XMLHttpRequest is for same-origin requests(cannot come from attacker)
  - Browser prevents sites from sending custom HTTP headers to other sites, but can send to themselves
- no secrets required

SameSite Cookie

attribute prevents cookie from being sent along with cross-domain requests

- strict flag
- lax flag: cookie sent even in case of cross-domain requests, but then must be change in top-level navigation

Broader view of CSRF

- abuse of cross-site data export
  - SOP does not control data export
  - malicious webpage can initiate requests from user's browser to honest server
  - server thinks requests are part of established session between browser and server
- many reasons for SCRF attacks, not just "session riding"
- recommendations:
  - use web frameworks that implement secret token method correctly
  - sameSime cookies

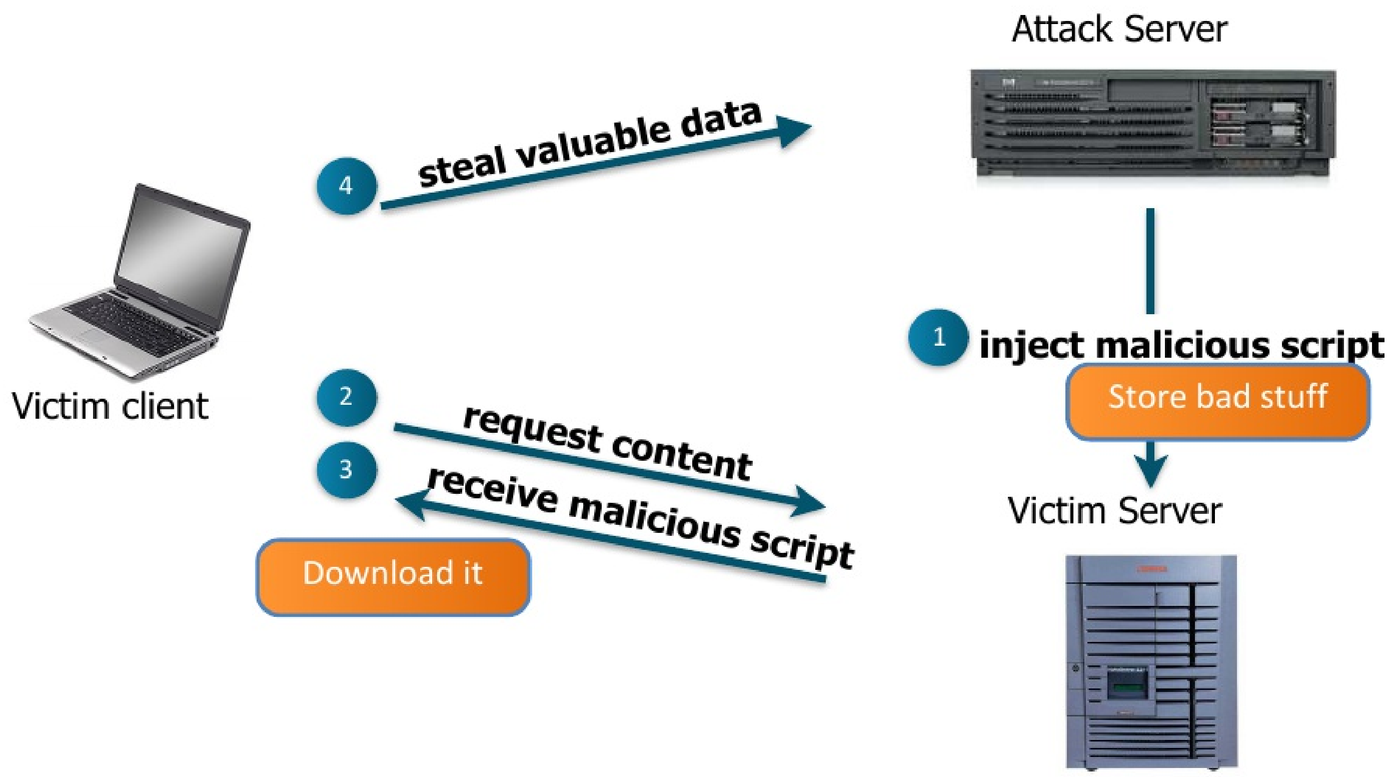
Cross Site Scripting (XSS)

- XSS vulnerability is present when attacker can inject scripting code into pages generated by web application.
- methods:
  - reflected XSS (type 1): attack script reflected back to user as part of page from victim site
  - stored XSS (type 2): attacker stores malicious code in resource managed by web application
  - others, such as DOM-based attacks

Reflected XSS

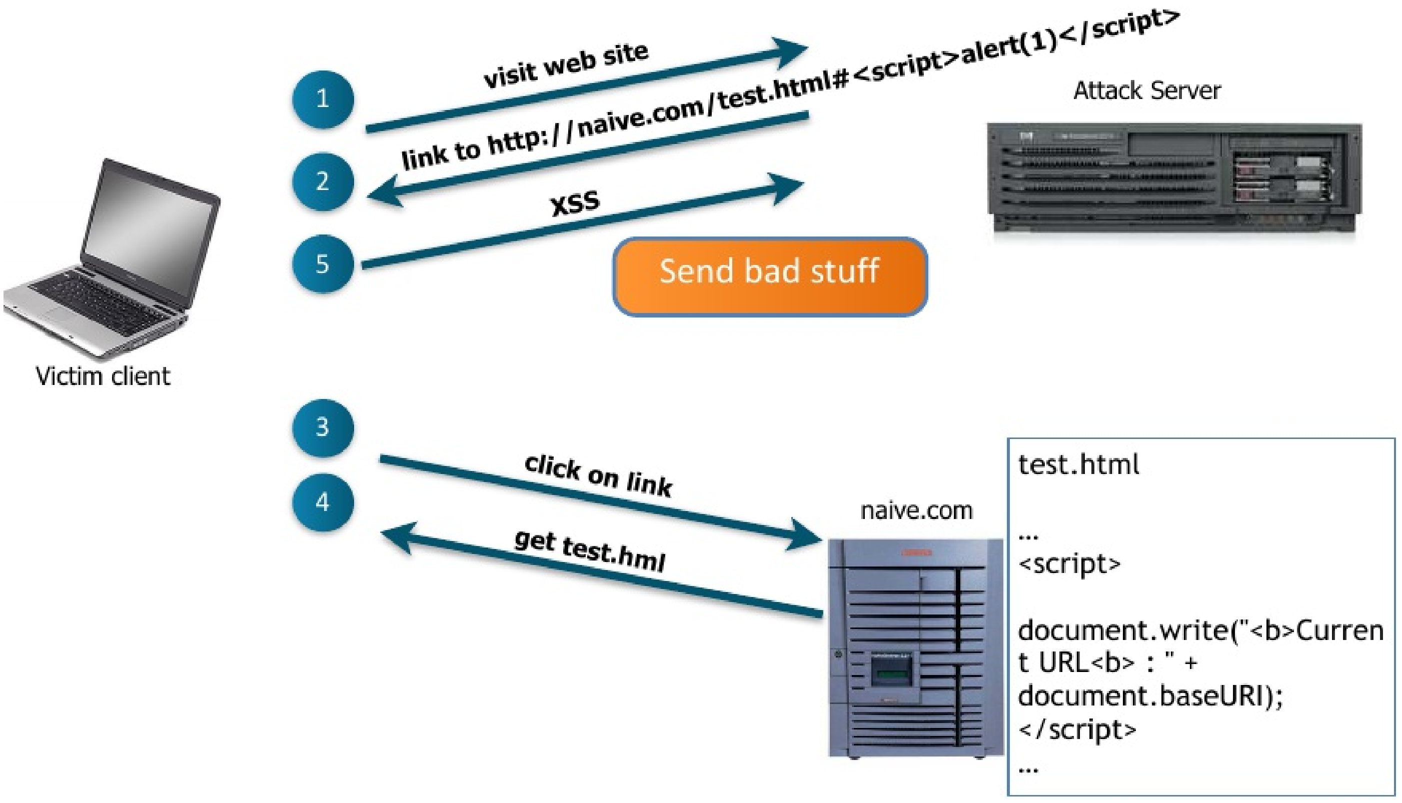
- user tricked into visiting honest website
  - phishing mail, link in banner ad, comment in blog
- bug in website code causes it to echo to user's browser an attack script
  - origin of script is now website itself
- script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields, ...
  - violates "spirit" of same origin policy, but not the letter
- where malicious scripts lurk:
  - user-created content
  - when visitor loads page, website displays content and visitor's browser executes script

Stored XSS



example: using "images", that are scripts

DOM-Based XSS



```
test.html

...
<script>

document.write("<b>Curren
t URL<b> : " +
document.baseURI);
</script>
...
```

- attacker's code is never sent to server - instead written by browser directly in the DOM, rendered there - server-side detection techniques don't work - problem: input from user goes to execution point

Defenses

Open Web Application Security Project (OWASP)

- ensure that app validates all headers, cookies, query strings, form fields, hidden fields against rigorous specification of what should be allowed

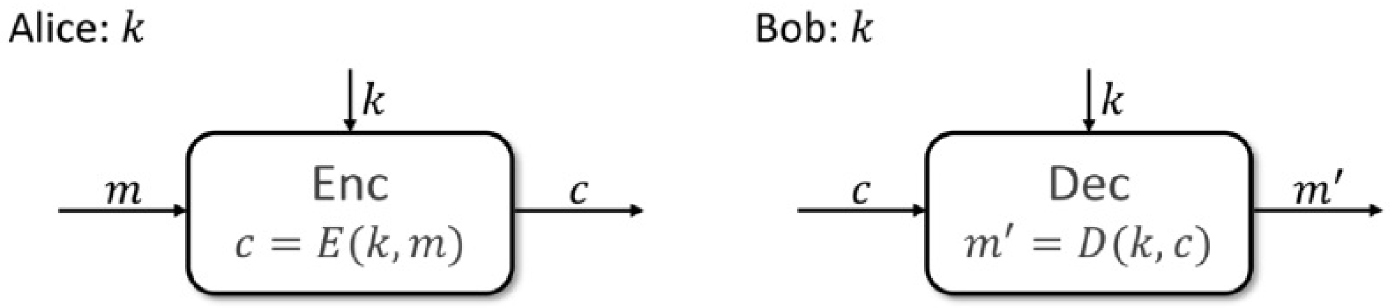
- do not attempt to identify active content and remove, filder, sanitize it. too many types of active content, too many ways of encoding to get around filters
- 'positive' security policy that specifies what is allowed is recommended
- 'negative' policies difficult to maintain, likely to be incomplete

preventing XSS

- any user input and client-side data must be preprocessed before usage inside HTML
- remove/encode (X)HTML special characters
- use good escaping library

Symmetric Cryptography

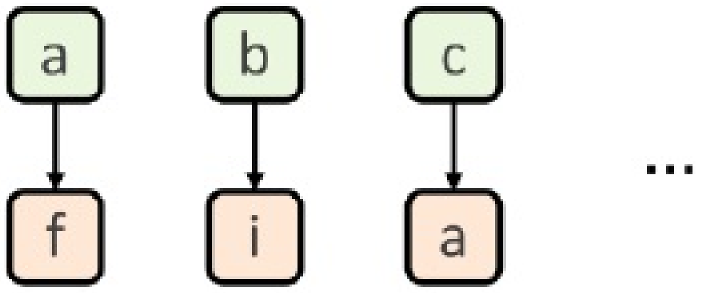
Ciphers



Symmetric encryption: both Alice and Bob use same key *k*

ancient ciphers

- substitution ciphers



  - easy to break: letter frequency analysis, frequency of pairs of letters, ciphertext only attack

Vigenere Cipher

- key randomly chosen string of certain length n
- easy to break (frequency analysis)

rotor machines

- key is initial position of rotor
- encryption/decryption by rotations, presumably hard to invert without knowing starting position
- with nowadays knowledge easy to break

Definition of Ciphers

A symmetric encryption scheme with key space *K*, message space *M*, ciphertext space *C* is a triple of algorithms (Gen, Enc, Dec):

- randomized key generation algorithm Gen takes no input, returns a key k∈*K*
- encryption algorithm Enc takes k and message m∈*M*, returns ciphertext c∈*C*
- deterministic decryption algorithm Dec takes k and ciphertext c, returns plaintext m∈*M* or distinguished error symbol

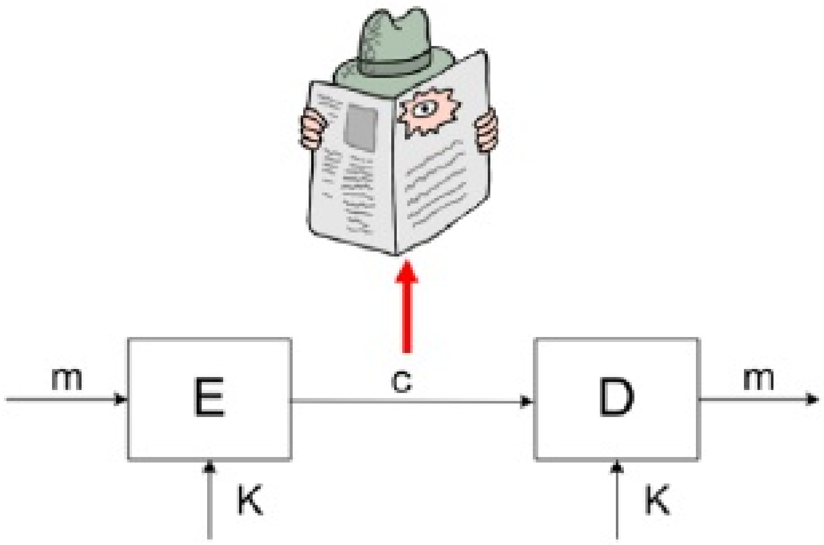
Types of Adversary Success

- **total break**: find the key
- **universal break**: find equivalent method to being able to decrypt with key
- **partial break**: successfully decrypt only selected ciphertexts, but those completely
- **partial information**: successfully learn partial information about single plaintext

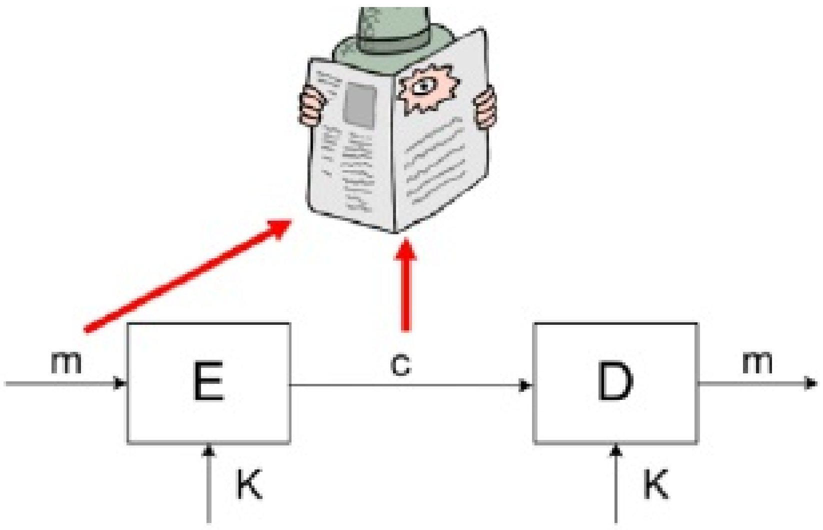
On Attack Models for Ciphers

Passive Attacks

- ciphertext only attack: observation of ciphertext

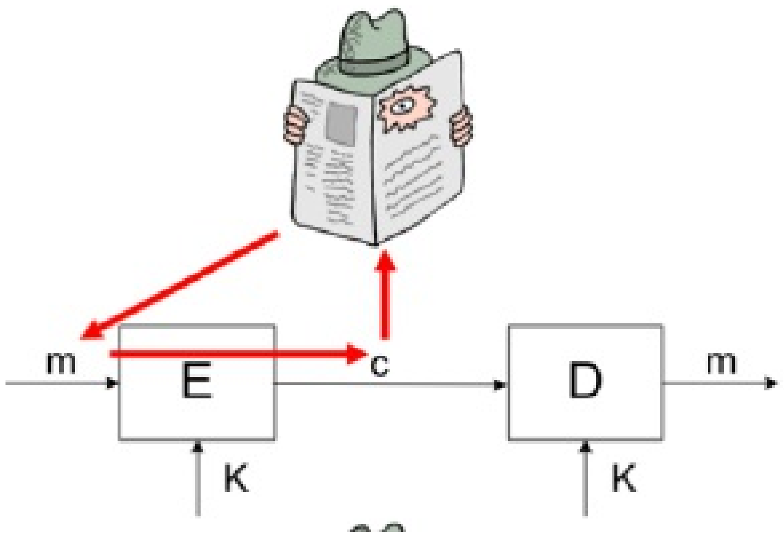


- known plaintext attack: observation of plaintext

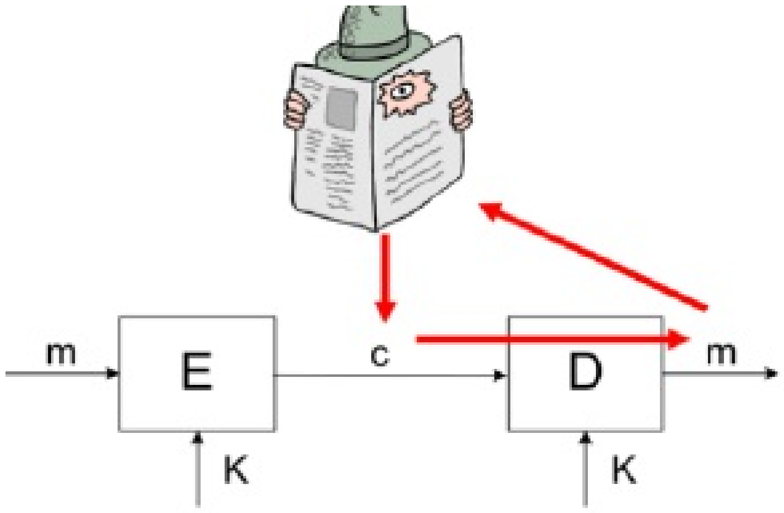


Active Attacks

- chosen plaintext attack: plaintext selectable



- chosen ciphertext attack: ciphertexts selectable



Security of Ciphers

- Basic idea: define that a ciphertext reveals **no** information about its plaintext
- convention: M, C, K random variables denoting value of message, ciphertext, key, respectively. consider only probability distributions that assign non-zero probability to all elements of respective space

One-time Pad (OTP)

- first proven-secure cipher
- secret key $k \in \{0,1\}^n$ = random bit string as long as the message
- very fast encryption and decryption
- problem: key is as long as message
- have to generate lots of randomness

Stream Ciphers

- key of OTP is random string $k \in \{0,1\}^n$
- ides: replace *random* by *pseudorandom*, secret key is now a seed
- PRG: pseudorandom generator expands small seed of random bits into large amount of somewhat random bits
- PRGs should be unpredictable
    - often prefix of message is known, e.g. fixed header of e-mail
    - lots of PRGs don't satisfy this

    - don't use UNIX rand for security
- not only about hiding seed, also not allowing to "look forward" which randomness will be created

getting true randomness

- weak randomness in practice:
    - throw coins, ...
    - derived from load/system parameters, ...
- stronger: exploit different physical processes that are expected to be random
    - thermal noise, air perturbation, ...
- randomness from all inputs XORed, hashed to remove bias

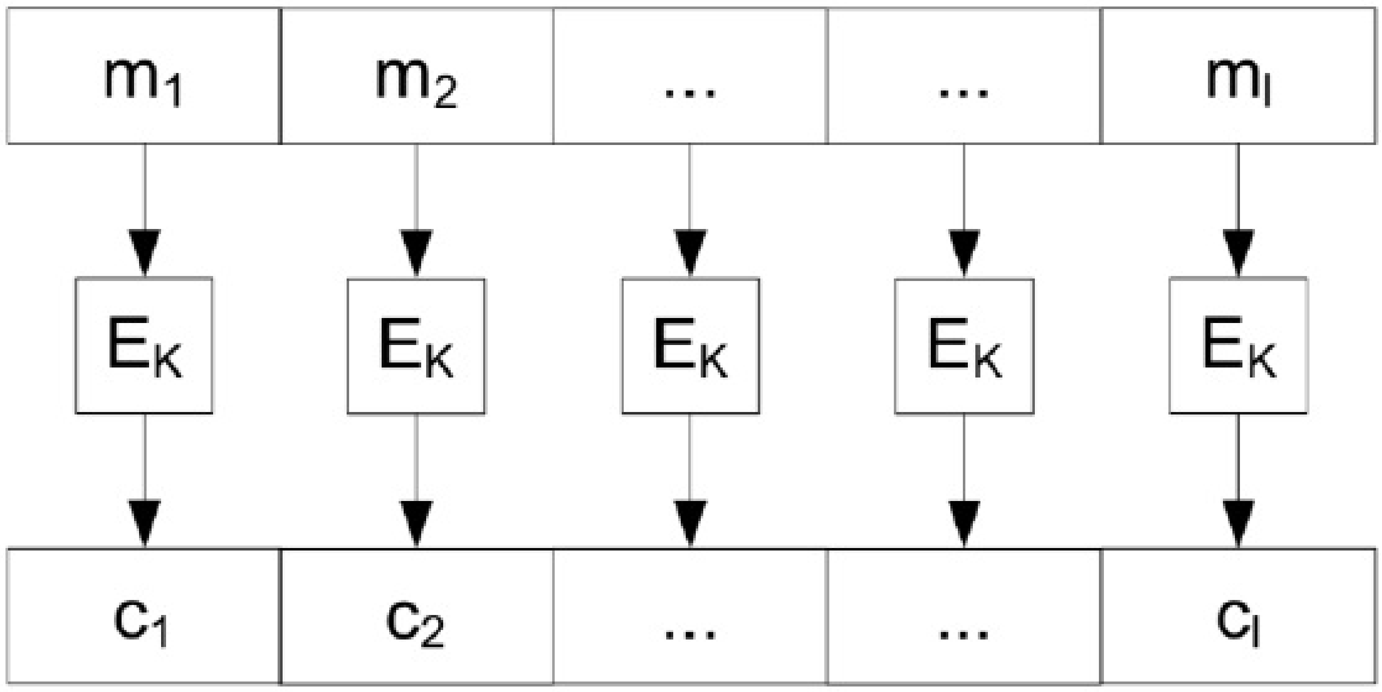
how to use stream cipher (for more than one message)

- assume we already have strong(but slow) sipher for encrypting small blocks and we have a PRG
    - pick random seed
    - transfer it in a secure way
    - use PRG to produce pseudorandom stream
    - use pseudorandom stream to encrypt message

Block Ciphers

deterministic algorithm operating on a fixed-length group of bits called block (as opposed to a stream of bits bit by bit)

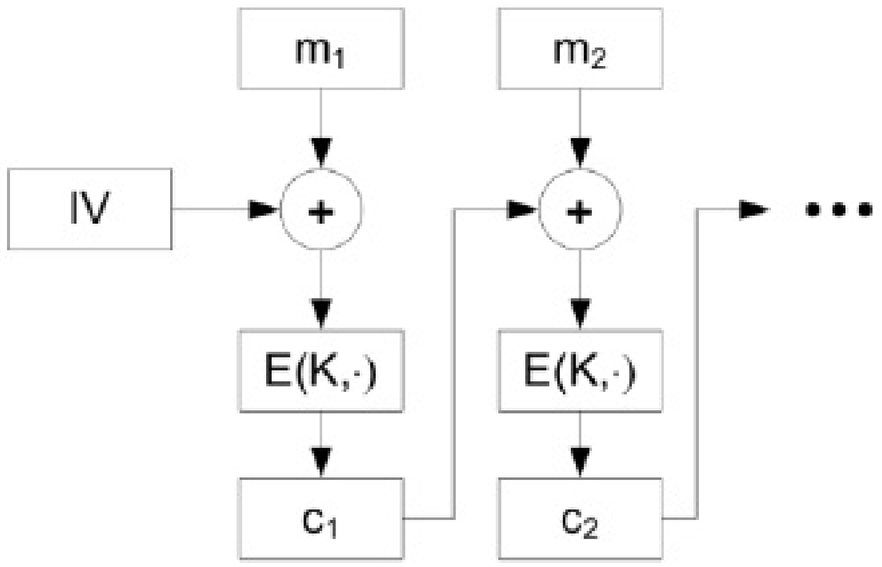
Electronic Codebook (ECB)



ECB reveals patterns

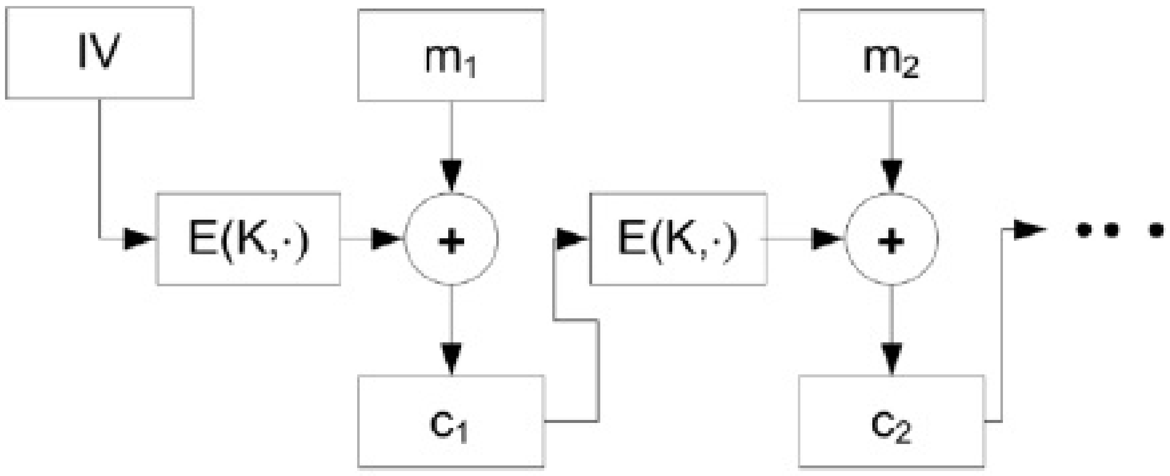
Cipherblock Chaining (CBC)

- initial value randomly chosen and output as well
- often used, but main problem: sequential
- self-synchronizing after two blocks (if block length ok)



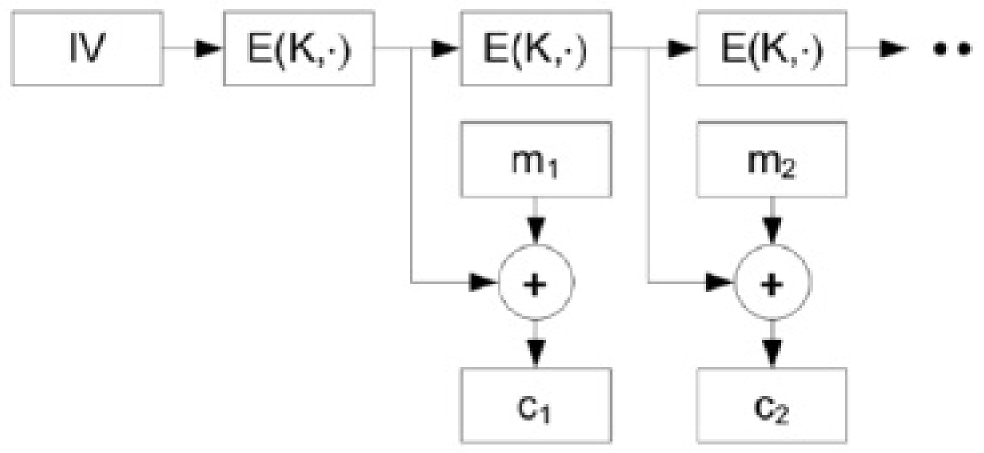
Cipher Feedback (CFB)

- similar to stream ciphers
- no need for decryption operation here
- encryption cannot be parallelized
- also self-synchronizing after two blocks



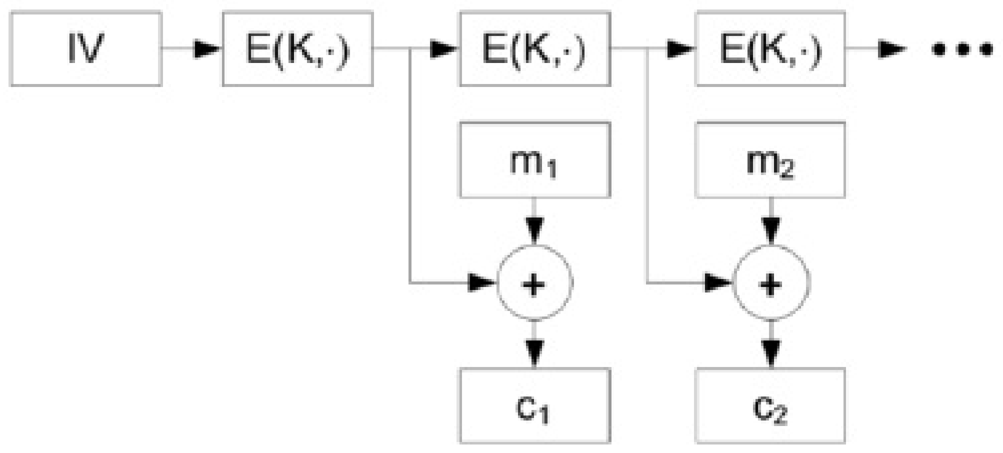
Output Feedback (OFB)

- similar to stream ciphers
- no need for decryption operation
- neither encryption nor decryption parallelizable
- stronly self-synchonizing

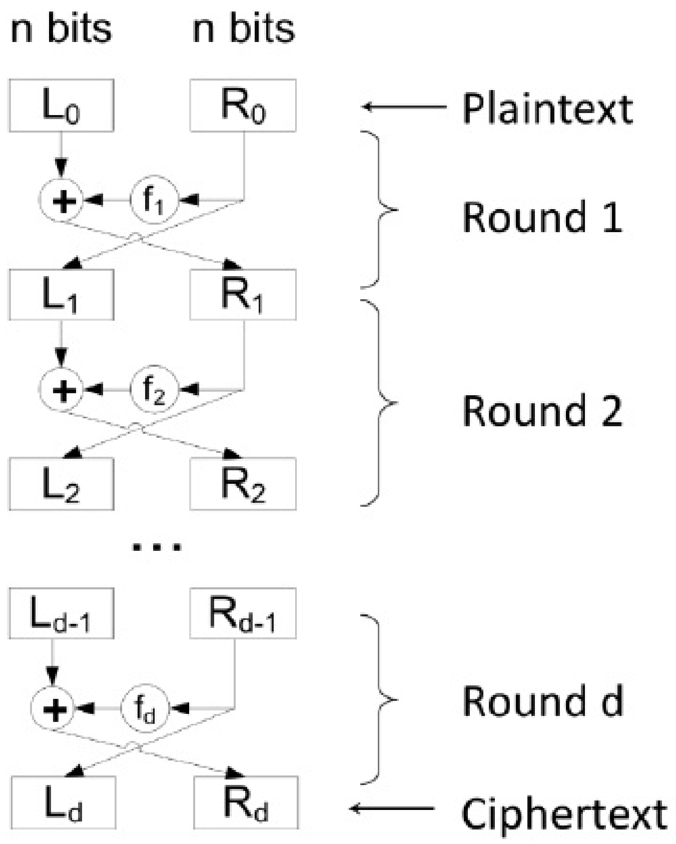


Countermode

- similar to stream ciphers
- no need for decryption operation
- ok to use same key for multiple messages if random IV, for fixed IV only one time key usage
- encryption and decryption parallelizable
- strongly self-synchonizing



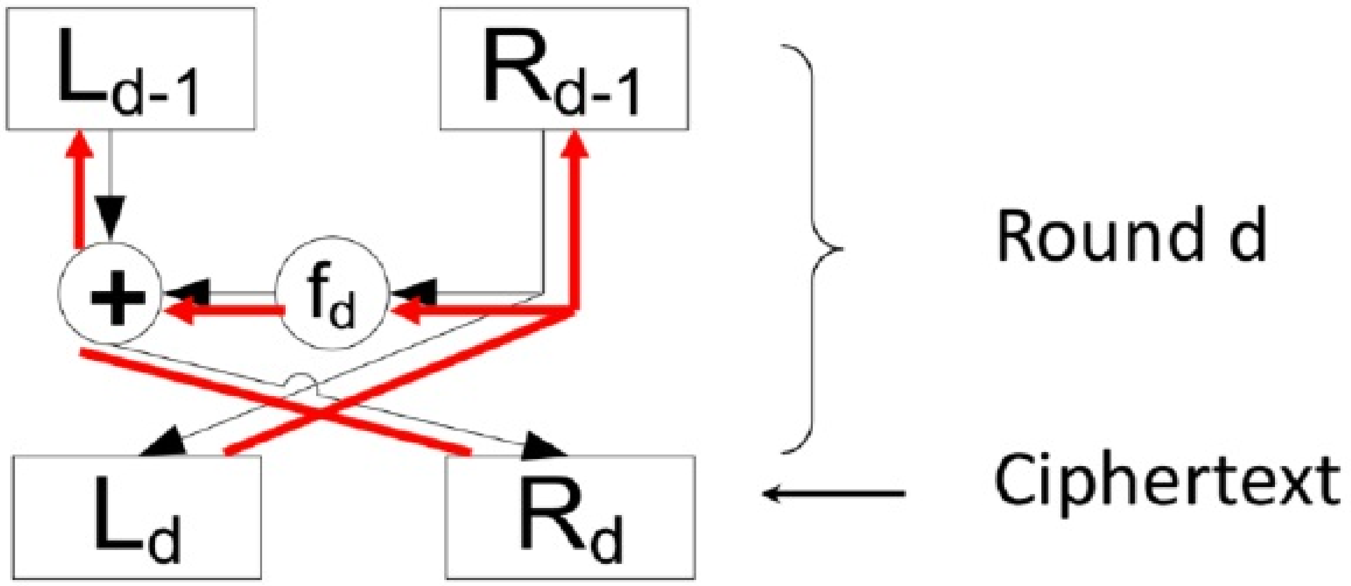
Data Encryption Standard (DES)



$f_1, \ldots, f_d : \{0,1\}^n \rightarrow \{0,1\}^n$ (for DES: n = 32 , d=16)

Feistel Networks

for any funcions *f1, ..., fd* a Feistel network is a one-to-one map



Feistel Network inverts itself

Exhaustive Search Attacks

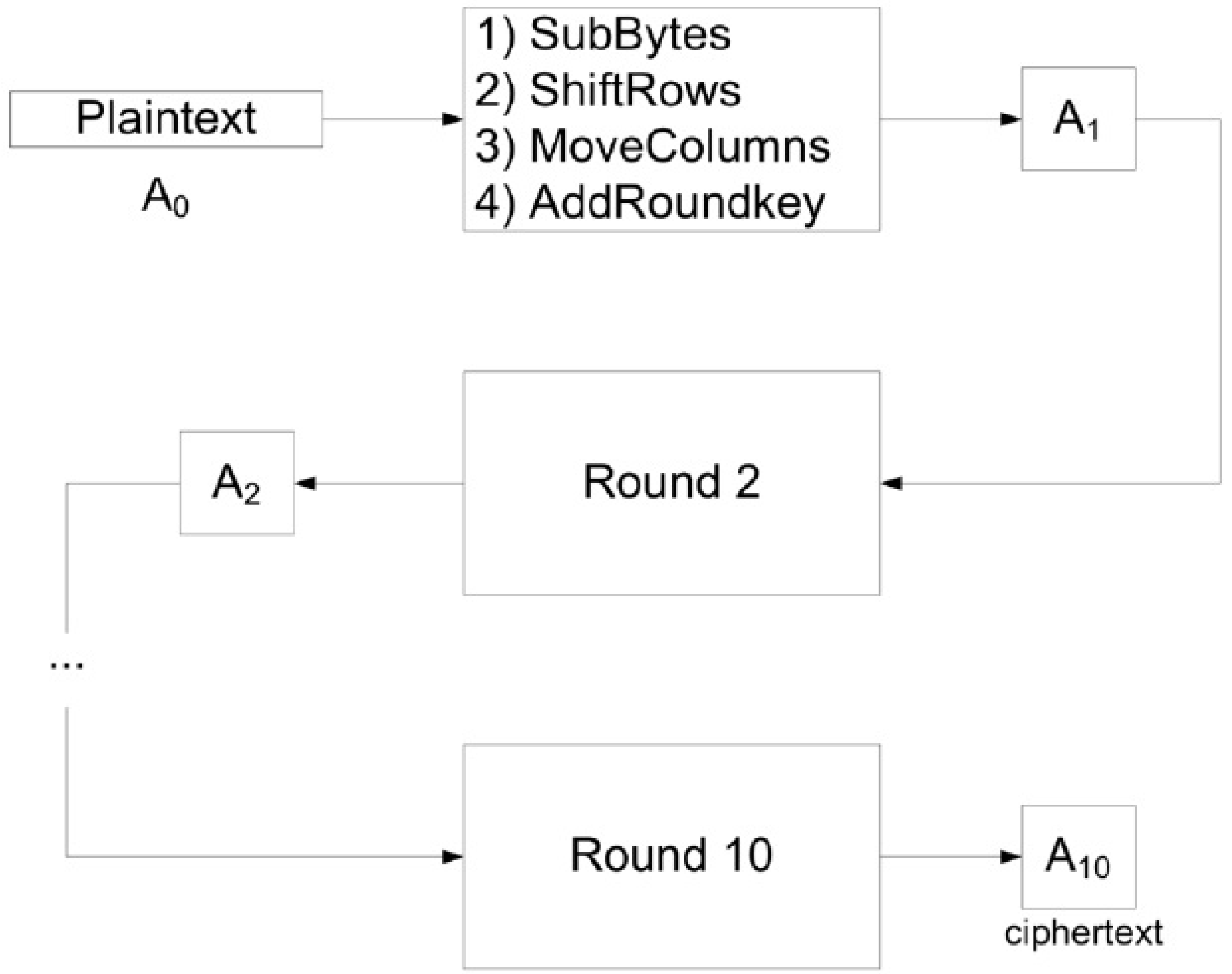
- most simple attack conceivable
- given: a few PT/CT pairs and random elements from $\{0, 1\}^n$
- goal: total break

- no stream ciphers would resist this setting: multiple encryptions with same key

AES

variant of Rindael

Rijndael Cipher



- AES round
  - SubBytes: A[i]←s-box(A[i,j])
  - ShiftRows: For i=0,1,2,3: Rotate left row i by i pos
  - MixColumns: Multiply each column to fixed matrix
  - AddRoundKey
  - subkey derived from main key using Rijndael's key schedule

MACs and Hashes

Message Integrity

- Alice generates tag t for message m, Bob verifies tag
- goal: attacker cannot change message

MACS

- Message Authentication Code

- message authentication code with message space *M*, key space *K*, tag space *T* is triple of algorithms (Gen, Sig, Ver):
  - randomized key generation algorithm Gen takes no input, returns key k
  - signing algorithm Sig takes k and message m, returns tag t
  - deterministic verification algorithm Ver takes k and m, returns bit b∈{0,1}

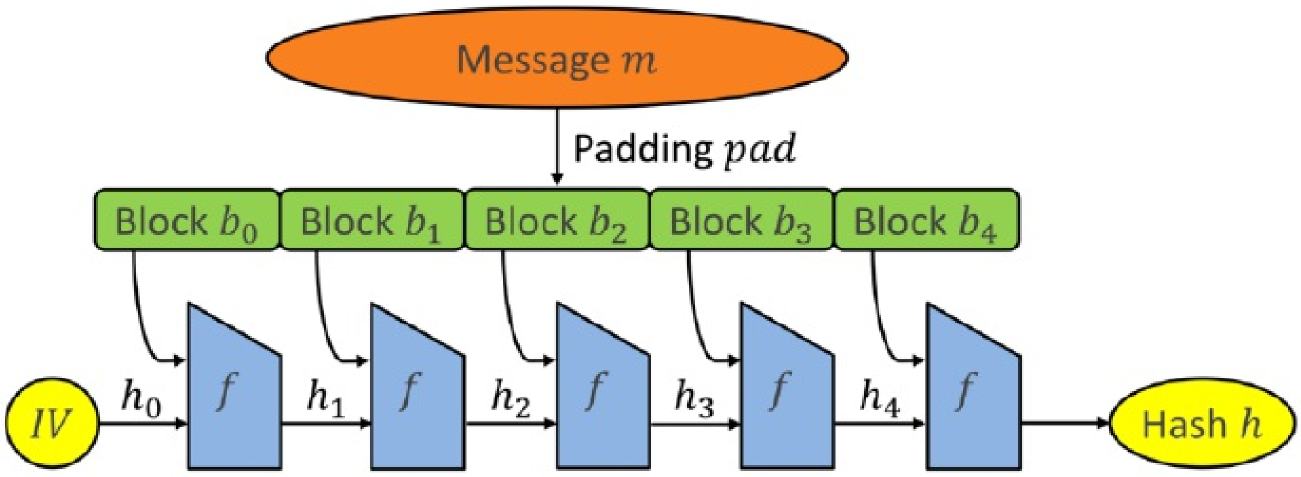
Hash Functions

- let *H*: *M*→*T* be hash fuction
- collision for *H* is tuple (m1, m2) with
  H(m1)=H(m2) ∧ m1!=m2
- Collision Resistant Hash Function (CRHF): hash function H is collision resistant if no "efficient" algorithm is known that finds a collision for H in suitable time
- Examples for CRHF:
  - SHA-2 family, e.g. SHA-256
  - Whirlpool (AES)
  - SHA-3 family

- Broken:
  - MD5
  - SHA1

Constructing CRHFs

- Merkle-Damgard (iterated construction)



- pad = padding function, encodes length of m in last block

- f = compression function

- h = chaining variables

- IV = initial value

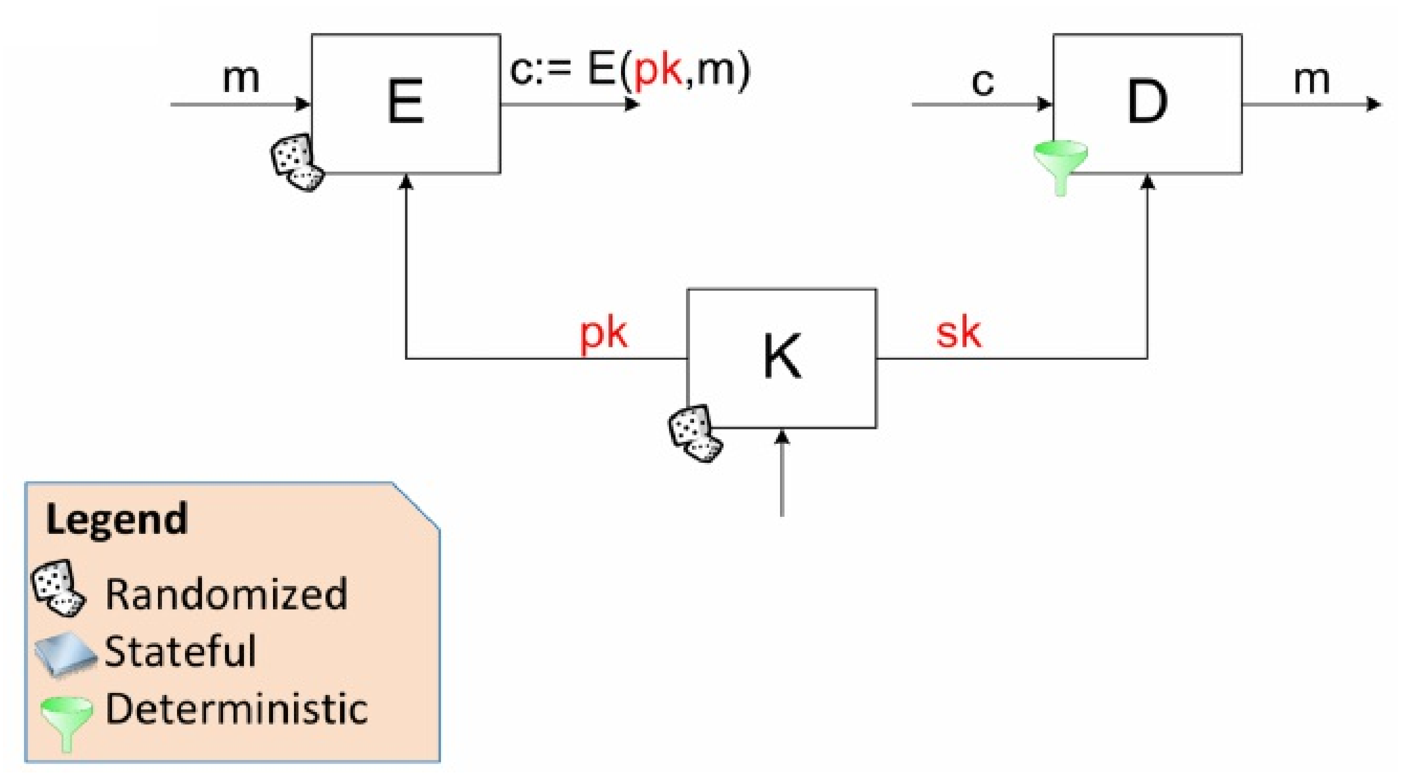
Asymetric Cryptography

- Symmetric Cryptography
  - fast (+)
  - based on heuristics (-)
  - one key for every pair of user (-)

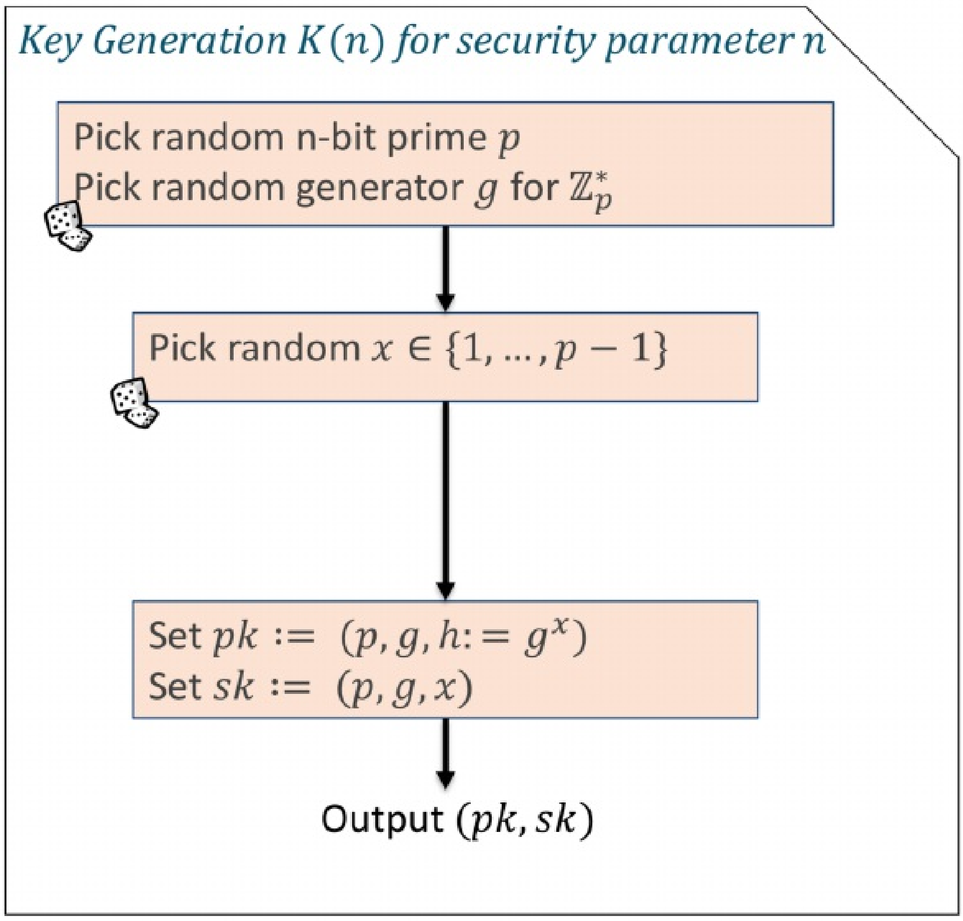
    - two parties need to protect the secret (-)

- Asymetric Cryptography
    - slow (-)
    - based on security proofs with well-defined assumptions (+)
    - one key for every user (+)
    - everyone is responsible for own secret key (+)

Public-Key Encryption



ElGamal Encryption Scheme



Key Generation $K(n)$ for security parameter n

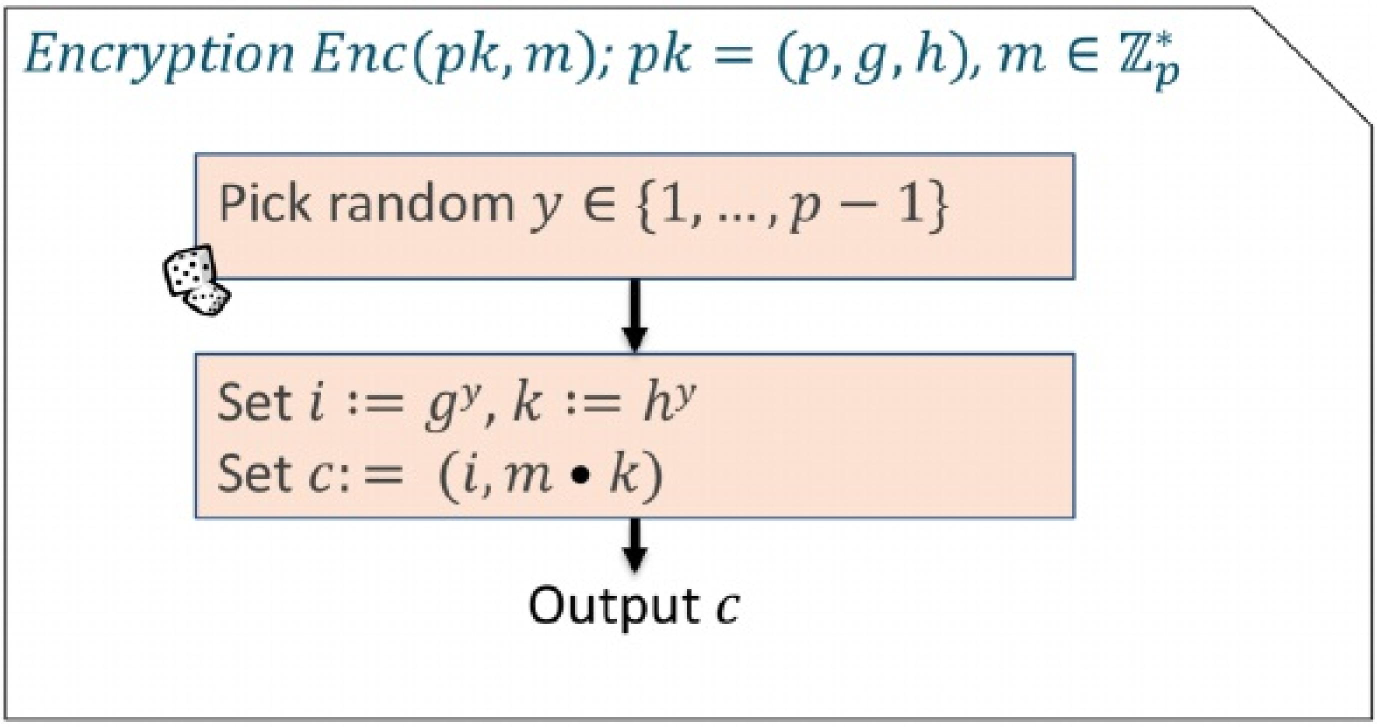
Pick random n-bit prime p
Pick random generator g for $\mathbb{Z}_p^*$

Pick random $x \in \{1, \dots, p-1\}$

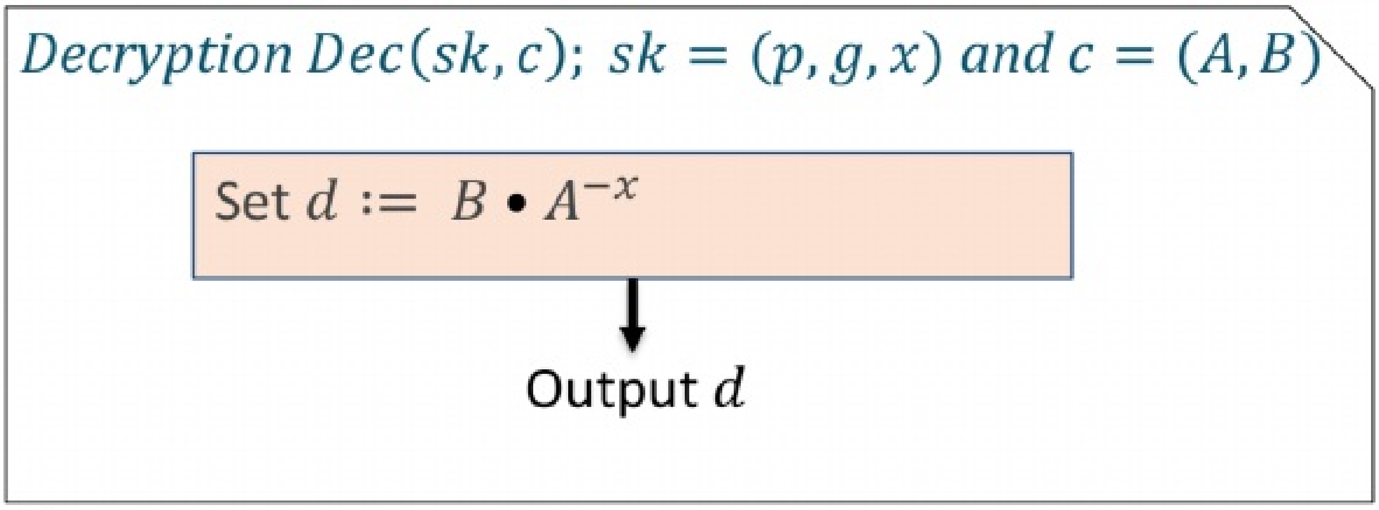
Set $pk := (p, g, h := g^x)$
Set $sk := (p, g, x)$

Output (pk, sk)

} Can be publicly known



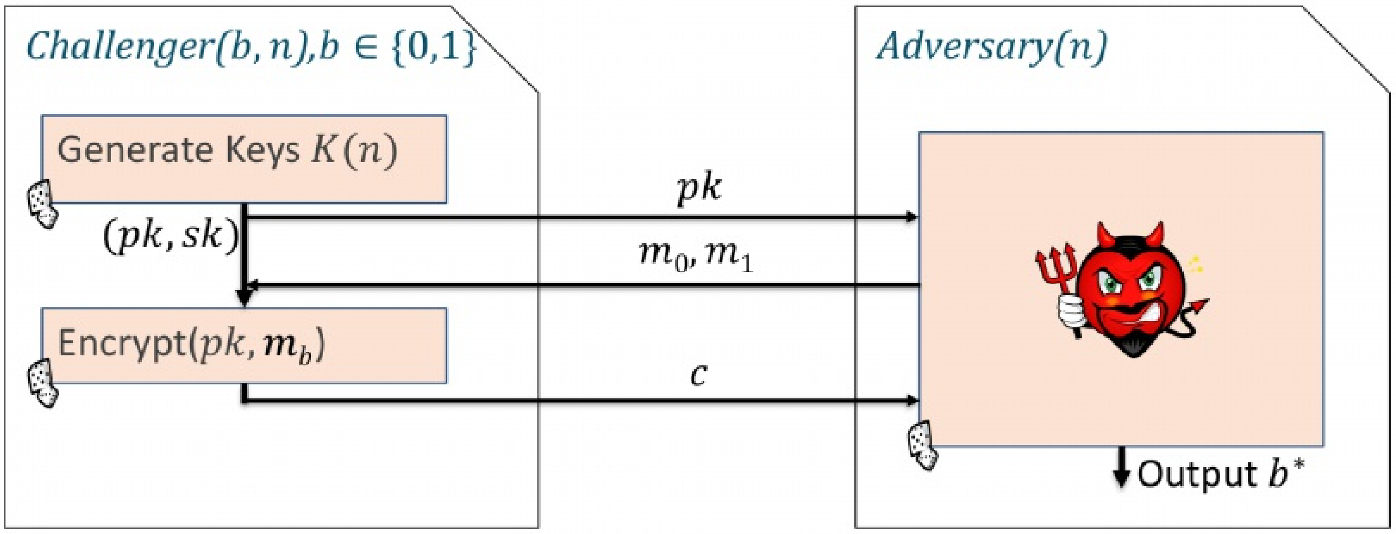
$Encryption\ Enc(pk, m);\ pk = (p, g, h),\ m \in \mathbb{Z}_p^*$
Pick random $y \in \{1, \dots, p-1\}$
Set $i := g^y, k := h^y$
Set $c := (i, m \bullet k)$
Output c



$Decryption\ Dec(sk, c);\ sk = (p, g, x)\ and\ c = (A, B)$
Set $d := B \bullet A^{-x}$
Output d

CPA-Security

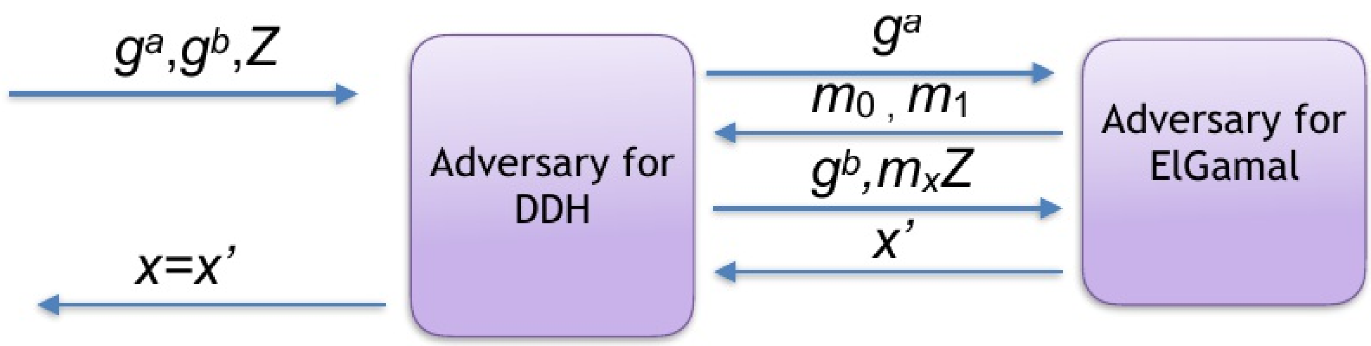
Let $PE = (K, E, D)$ be a public-key encryption scheme and A an adversary. Define $Exp_{PE,A}^{CPA}(b)$ as:



ElGamal → DDH

Decisional Diffie-Hellman Assumption (DDH)

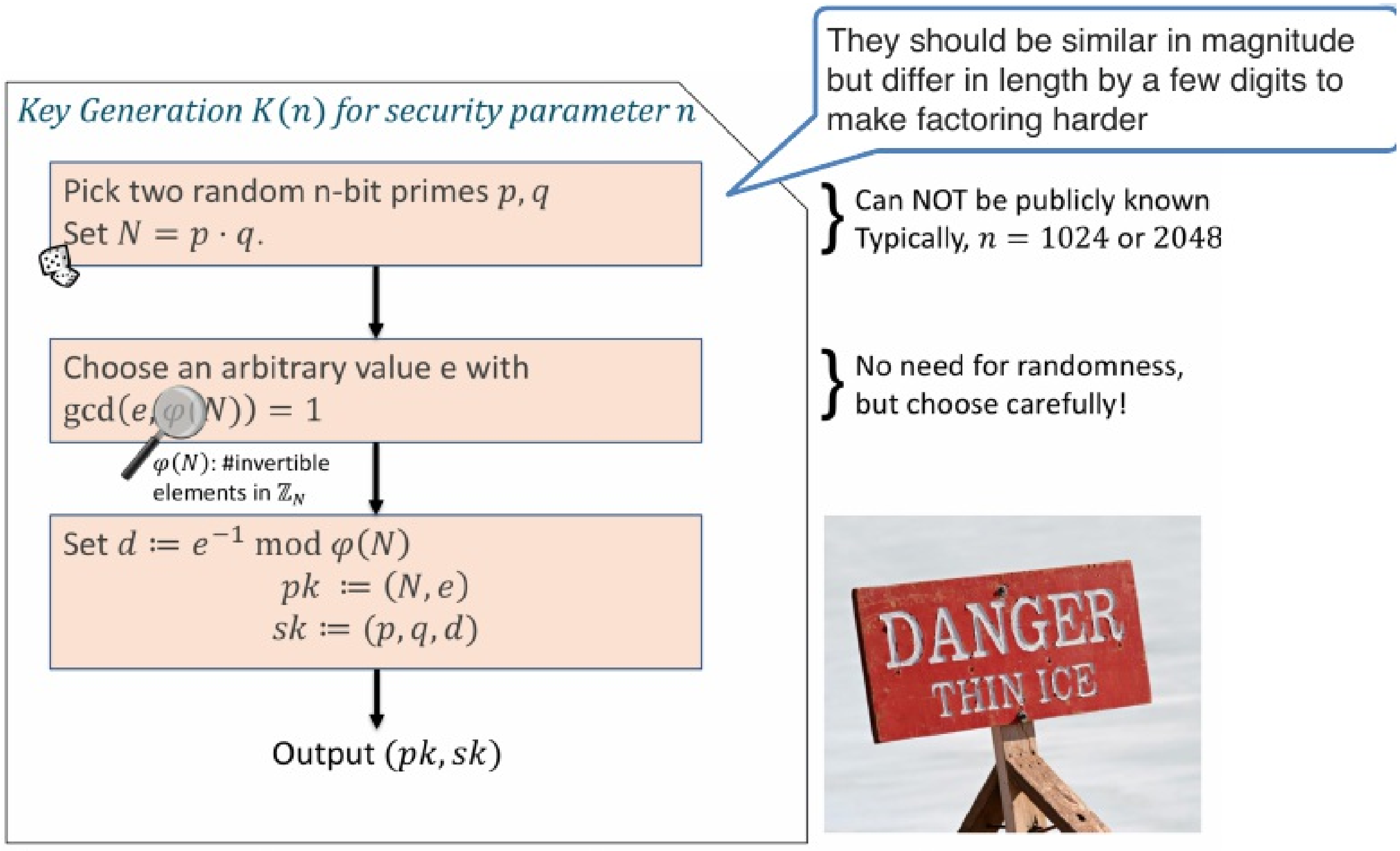
Given a group G with ~2^n elements and random g∈G, no polynomially adversary (in n) can distiguish
(g^x, g^y, g^{xy}) and (g^x, g^y, g^{xy})
for x, y, z random in {i,...,|G|}



- suppose some algorithm A breaks ElGamal
- we will use A to break DDH

RSA

RSA Trapdoor Permutation



Key Generation $K(n)$ for security parameter n

Pick two random n-bit primes p, q
Set $N = p \cdot q$.

Choose an arbitrary value e with
$\gcd(e, \varphi(N)) = 1$

$\varphi(N)$: #invertible elements in $\mathbb{Z}_N$

Set $d := e^{-1} \bmod \varphi(N)$
$pk := (N, e)$
$sk := (p, q, d)$

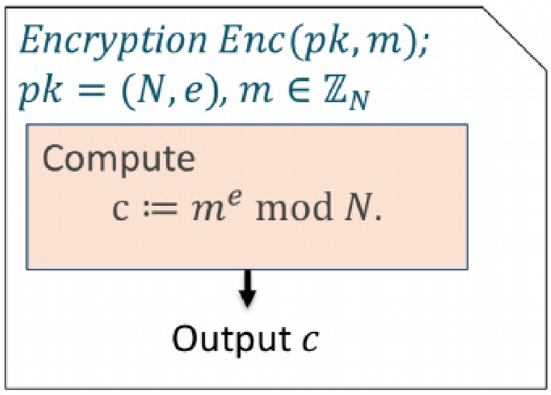
Output (pk, sk)

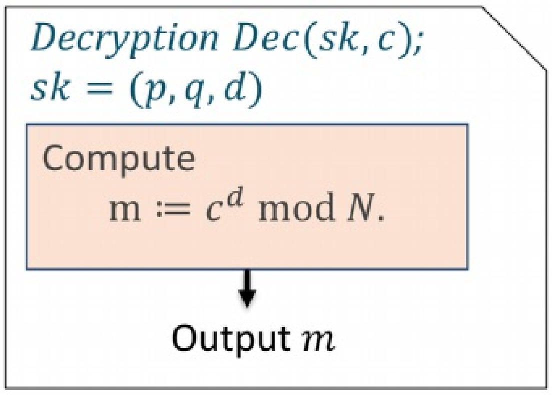
They should be similar in magnitude but differ in length by a few digits to make factoring harder

} Can NOT be publicly known
Typically, $n = 1024$ or 2048

} No need for randomness, but choose carefully!

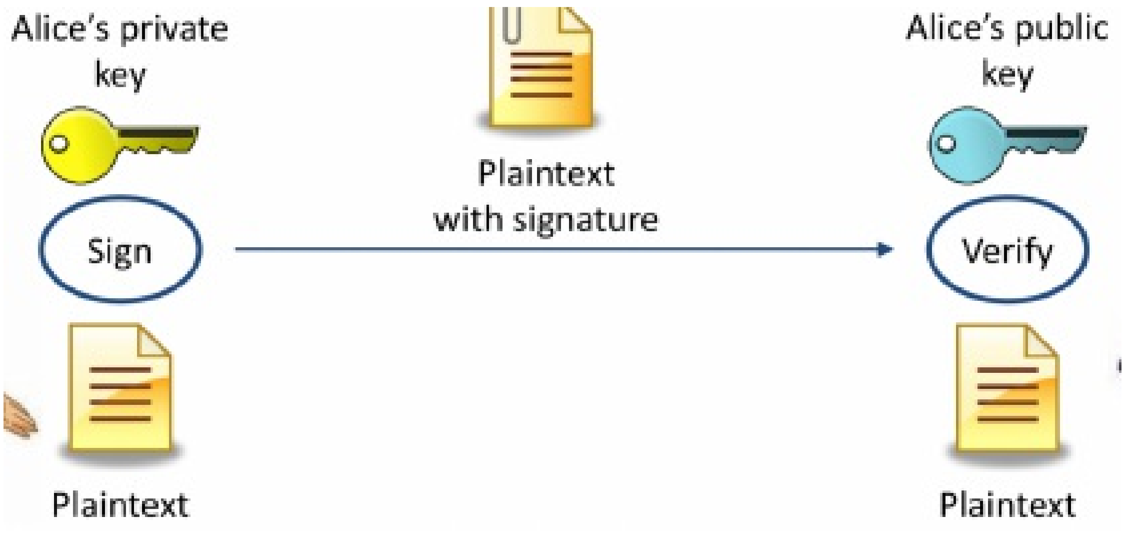
Naive use of RSA scheme





Problem: not even secure against passive attacks

Digital Signatures



- only the secret key allows for creating signatures
- everybody can verify validity of signatures using respective public key
- signatures serve as undisputable evidence that respective person signed message

Digital signatures vs MACs

- MACs provide integrity: why need to digital signature?
- public verifiability: everybody can verify signature, integrity proofs can be transferred
- non-reputation: signer cannot deny having signed document
  - with MAXs, key is secret
  - even if revealed to judge, judge does not know whether it is real or faked a posteriori
  - even if judge knows somehow right key, judge does not know which of two parties has signed document

Naive RSA-based Signatures

Key Generation as for RSA encryption for primes p, q:

Set $N \coloneqq pq$

↓

Pick random e, with
$1 \leq e \leq \phi(N)$ and
$\gcd\big(e, \phi(N)\big) = 1$

↓

Set $d \equiv e^{-1} \bmod \phi(N)$

↓

Set $pk \coloneqq (N, e)$
Set $sk \coloneqq d$

↓

Output (pk, sk)

} Can be publicly known

Signing $S(sk, m)$

Set $t \coloneqq m^d \bmod N$

↓

Output t

Verifying $V(pk, m, t)$

Test if $t^e \equiv m \bmod N$

↓

Output $b \in \{0,1\}$

Cryptographic Protocols

- cryptographic protocols building block of
    - e-banking
    - e-commerce
    - e-mail
    - e-voting
    - e-passports
    - online auctions
    - file sharing
    - social networks
- tons of attacks
    - conceptual flaws in protocol design
    - cryptographic breaches
    - implementation mistakes
- flaws hard to spot, proofs hard to get right

an untrusted world

- trusted principals exchange messages on a network populated by malicious entities
- everybody can read/write messages in transit on network

cryptography

- k is symmetric key shared between A and B: only A and B can encrypt/decrypt messages with k
- symmetric cryptography protects secrecy and integrity of message
    - enemy cannot read or modify transfer request
- cryptography is not enough
- attacker can circumvent cryptography and break security goals of protocol by intercepting, duplicating, sending back messages in transit on network, without need to break encryption scheme

safe communication

- for securing communication we need to
    - identify security goals
    - determine threat model
    - protect messages in transit on network accordingly
- Security Goals:
    - Secrecy: only authorized recipient should be able to learn message
    - Integrity: recipient should be able to determine whether message has been altered during transmission or not
    - Authenticity:
        - non-injective agreement: recipient of authentication request should be able to verify the identity of requester, both should agree on respective roles
        - injective agreement: same as above + recipient should be able to verify freshness of authentication request

Attacker threats

attacks often surprising, hard to predict, but can be roughly classified according to kind of interaction between attacker and protocol sessions

Interleaving attack

- message generated in a protocol session is exploited by attacker to interact with an other simultaneausly ongoing session
- special type of interleaving attack is called reflection attack

Reflection attack

- interleaving attack in which message is sent back to generator
  - works when victim is playing multiple roles in protocol, possibly in different sessions
- attacker intercepts transfer requests, sends it back to A who recognizes message as generated by B, performs the transfer
- symmetric nature of encryption key k does not allow A to verify whether ciphertext has been generated by B or herself
- avoiding:
  - solution is to break symmetry of cryptographic scheme by inserting originator's identifier (or intended receiver)

Replay attack

- same message is duplicated, sent several times to intended recipient
- solutions:
  - insert timestamp for guaranteeing freshness of message
  - exploit a challenge-response nonce handshake
    - nonce = randomly generated number n, used in single protocol session, then discarded

Challenge-response handshakes

- challenge-response nonce handshakes common in cryptographic protocols, different ways
  - PC (plain-cipher) handshakes: challenge is in clear, response is encrypted
  - CP (cipher-plain) handshakes: challenge encrypted, response in clear
  - CC (cipher-cipher) handshakes: both encrypted
- common idea: principals prove identities by encrypting/decrypting challenge and response

PC handshake

- also known as ISO two-pass unilateral authentication protocol
- which form of authentication is guaranteed?
  - injective agreement (A authenticates with B)
- response might be signed with A's private key, identifier A replaced by B
  - otherwise receiver is not specified

CP handshake

- B authenticates A receiving message m
  - only A can decrypt
  - second message may be seen as receipt acknowlegment
- which form of authentication is guaranteed?
  - non-injective agreement (B authenticates with A)

    - injective agreement (A auth. with B)

- challenge might be encrypted by A's public key (identifier A replaced by B)
- non-injective agreement would not be guaranteed, since challenge might originate from attacker

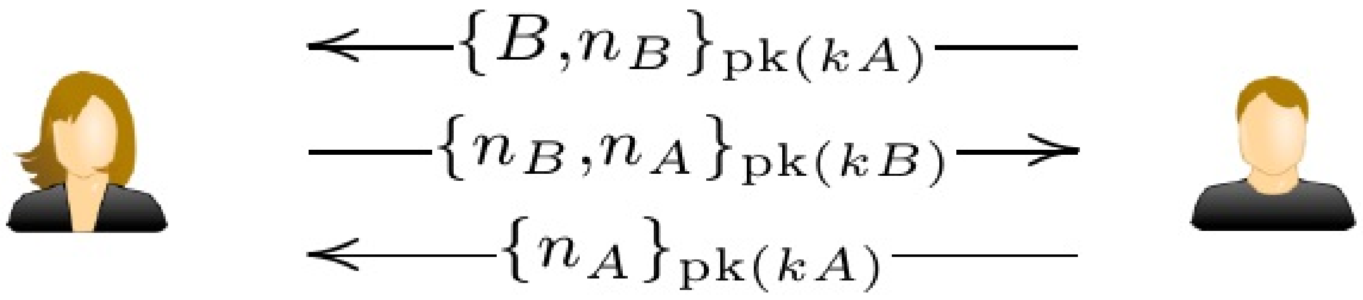
CC handshake

- B authenticates A receiving message m1 and sending message m2
- which form of authentication is guaranteed?
    - non-injective
    - injective

- can B be replaced with A in first message? (or A with B in second one)?
    - no, otherwise there would be an reflection attack

- challenge might be encryptet with A's public key, response encrypted with B's public key
- non-injective agreement would not be guaranteed, since challenge might originate from attacker

Needham-Schroeder-Protocol

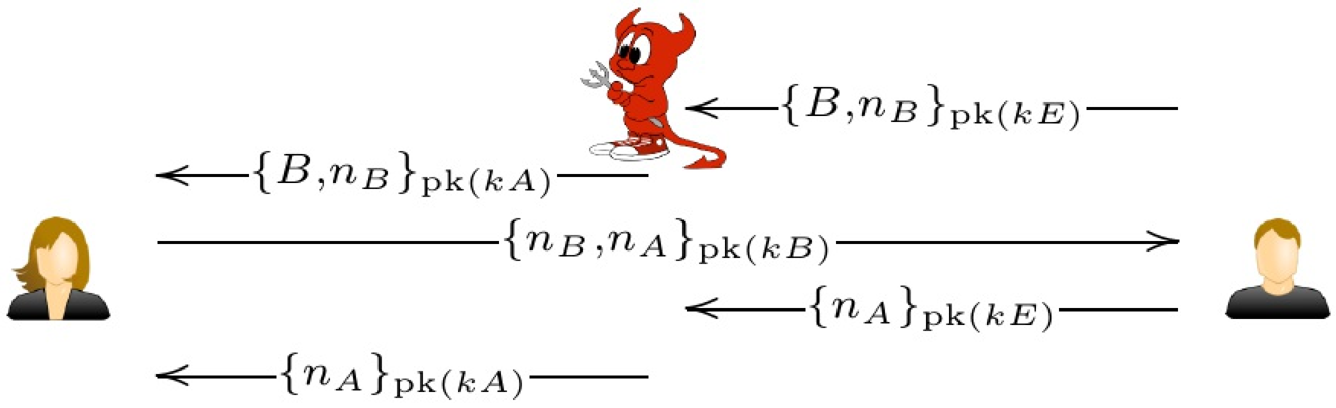


- pk(*kA*), pk(*kB*) public keys

- aim is to guarantee secrecy/Authenticity of two nonces, which are then used for generating symmetric session-key shared between A and B

- all messages are encrypted, two CC handshakes

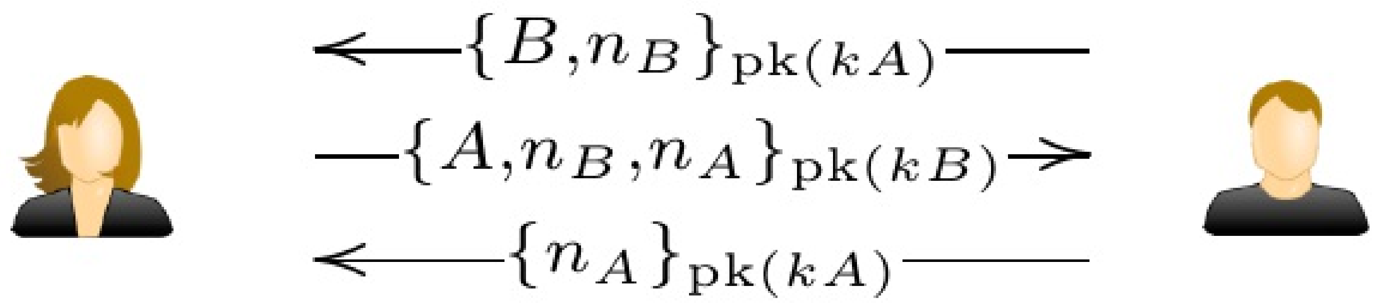
- protocol is not secure



    - A believes that B is authenticating with her, while B is authenticating with E
    - E learns two nonces, can build session key that A uses to talk with B

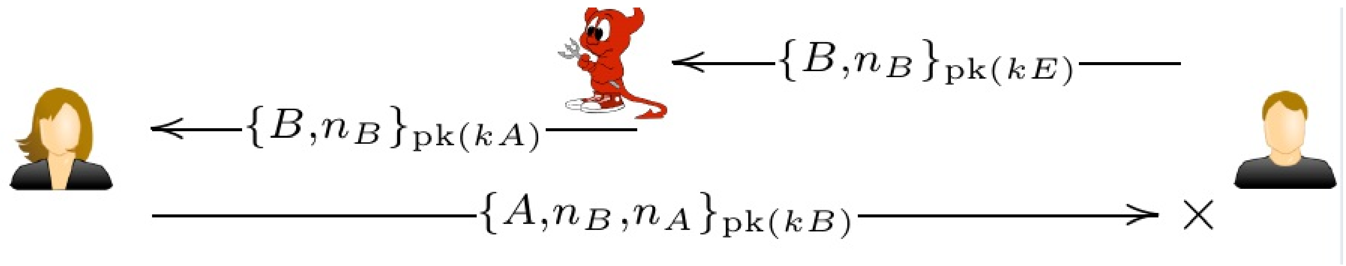
- man-in-the-middle-attack (MITM): adversary E steps into communication path, relays messages between legitimate parties A and B, itself acting as part of communication

Needham-Schroeder-Lowe Protocol



$\{B,n_B\}_{\mathrm{pk}(kA)}$
$\{A,n_B,n_A\}_{\mathrm{pk}(kB)}$
$\{n_A\}_{\mathrm{pk}(kA)}$

- fix consists in adding A's identifier in second ciphertext



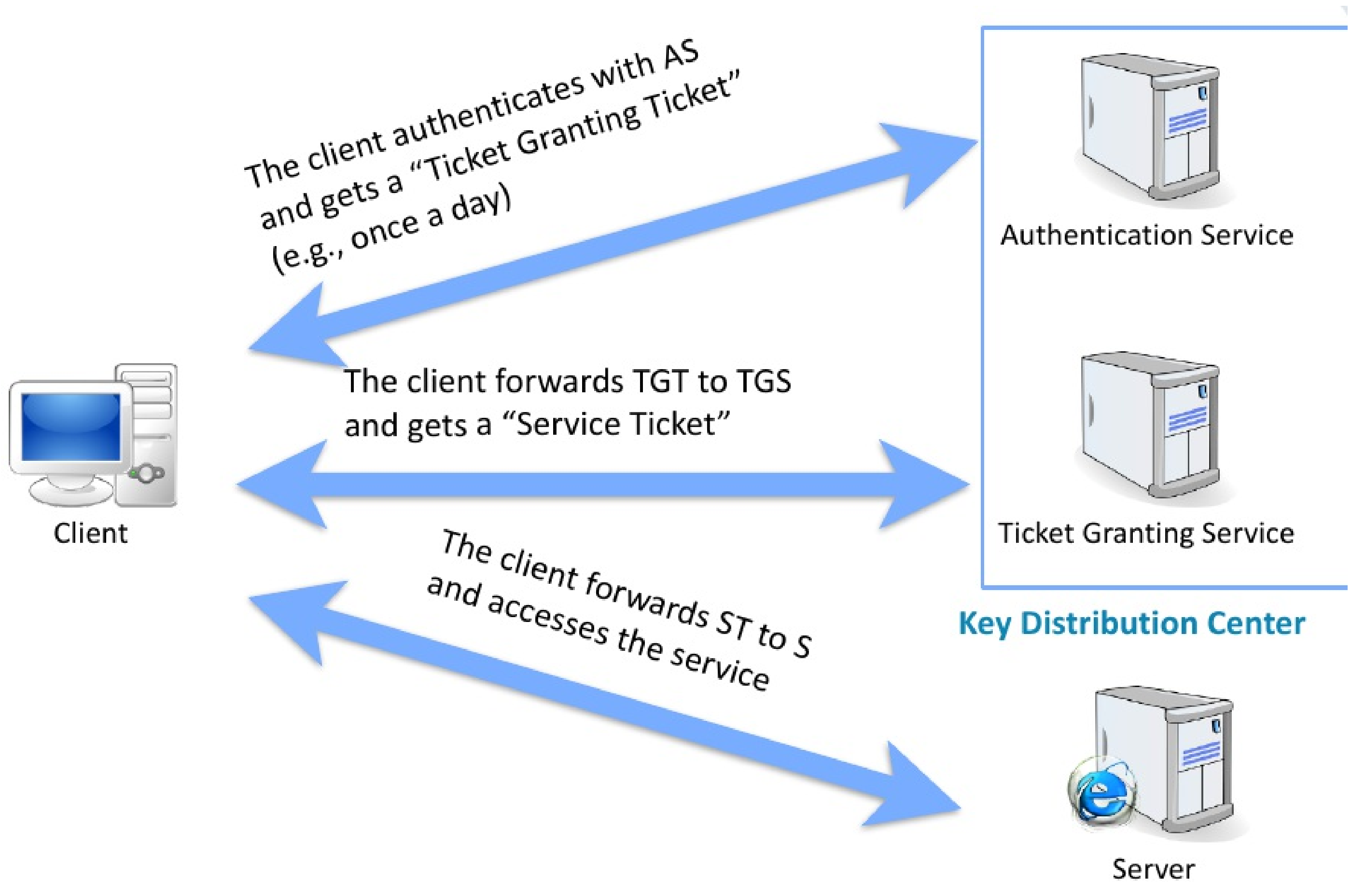
$\{B,n_B\}_{\mathrm{pk}(kE)}$
$\{B,n_B\}_{\mathrm{pk}(kA)}$
$\{A,n_B,n_A\}_{\mathrm{pk}(kB)}$

- B rejects second ciphertext, as it does not come from E

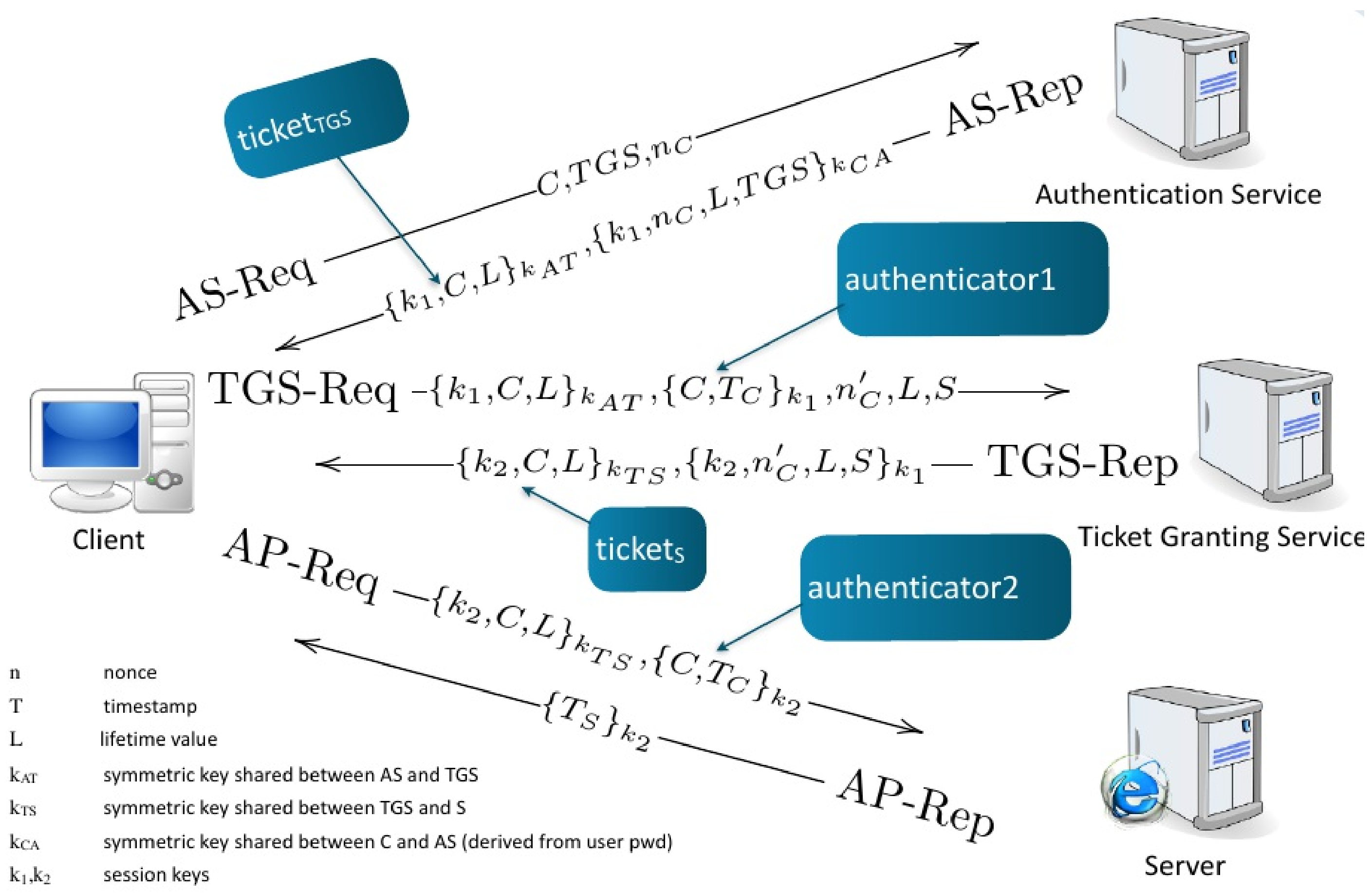
Prudent Engineering Practice

- **Principle 1:** every message should say what it means: interpretation of message should depend only on content. it should be possible to write down straightforward english sentence describing content, though if there is suitable formalism available that is good too
- **principle 3:** if identity of principal is essential to meaning of a message, prudent to mention principal's name explicitly in message

Kerberos



The client authenticates with AS and gets a "Ticket Granting Ticket" (e.g., once a day)
Authentication Service
The client forwards TGT to TGS and gets a "Service Ticket"
Client
Ticket Granting Service
The client forwards ST to S and accesses the service
Key Distribution Center
Server


ticket_TGS
AS-Req — C,TGS,n_C
{k_1,C,L}_kAT,{k_1,n_C,L,TGS}_kCA — AS-Rep
Authentication Service
authenticator1
Client
TGS-Req –{k_1,C,L}_kAT,{C,T_C}_k1,n'_C,L,S
{k_2,C,L}_kTS,{k_2,n'_C,L,S}_k1 — TGS-Rep
Ticket Granting Service
ticket_S
authenticator2
AP-Req —{k_2,C,L}_kTS,{C,T_C}_k2
{T_S}_k2 — AP-Rep
Server
n nonce
T timestamp
L lifetime value
k_AT symmetric key shared between AS and TGS
k_TS symmetric key shared between TGS and S
k_CA symmetric key shared between C and AS (derived from user pwd)
k_1,k_2 session keys

SSL/TLS

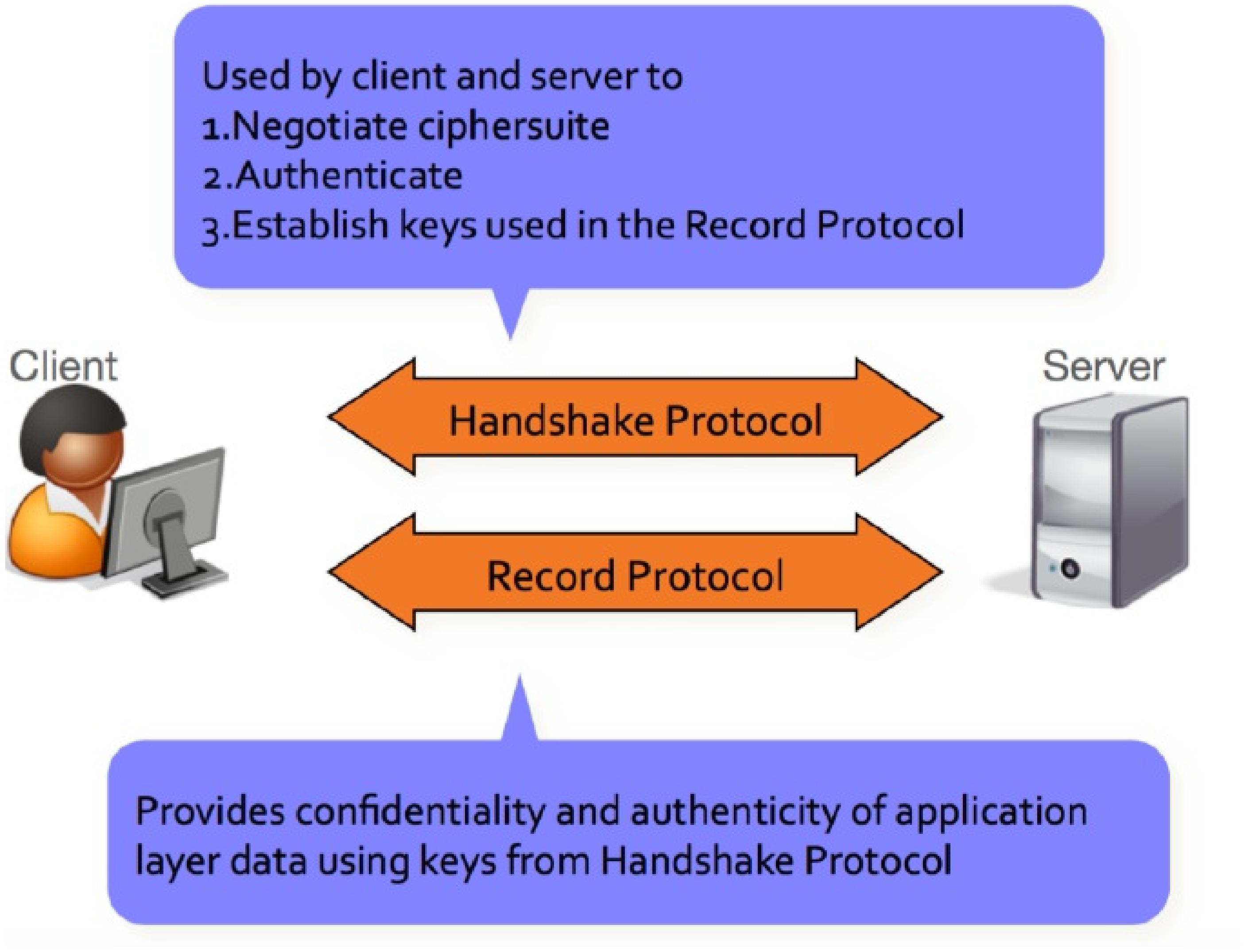
- Secure Sockets Layer and Transport Layer Security protocols
  - similar protocol design, different crypto algorithms
- De facto standard for Internet security
- deployed in every web browser, also VoIP, payment systems, distributed systems, etc.
- End-to-end secure communications in presence of a network attacker

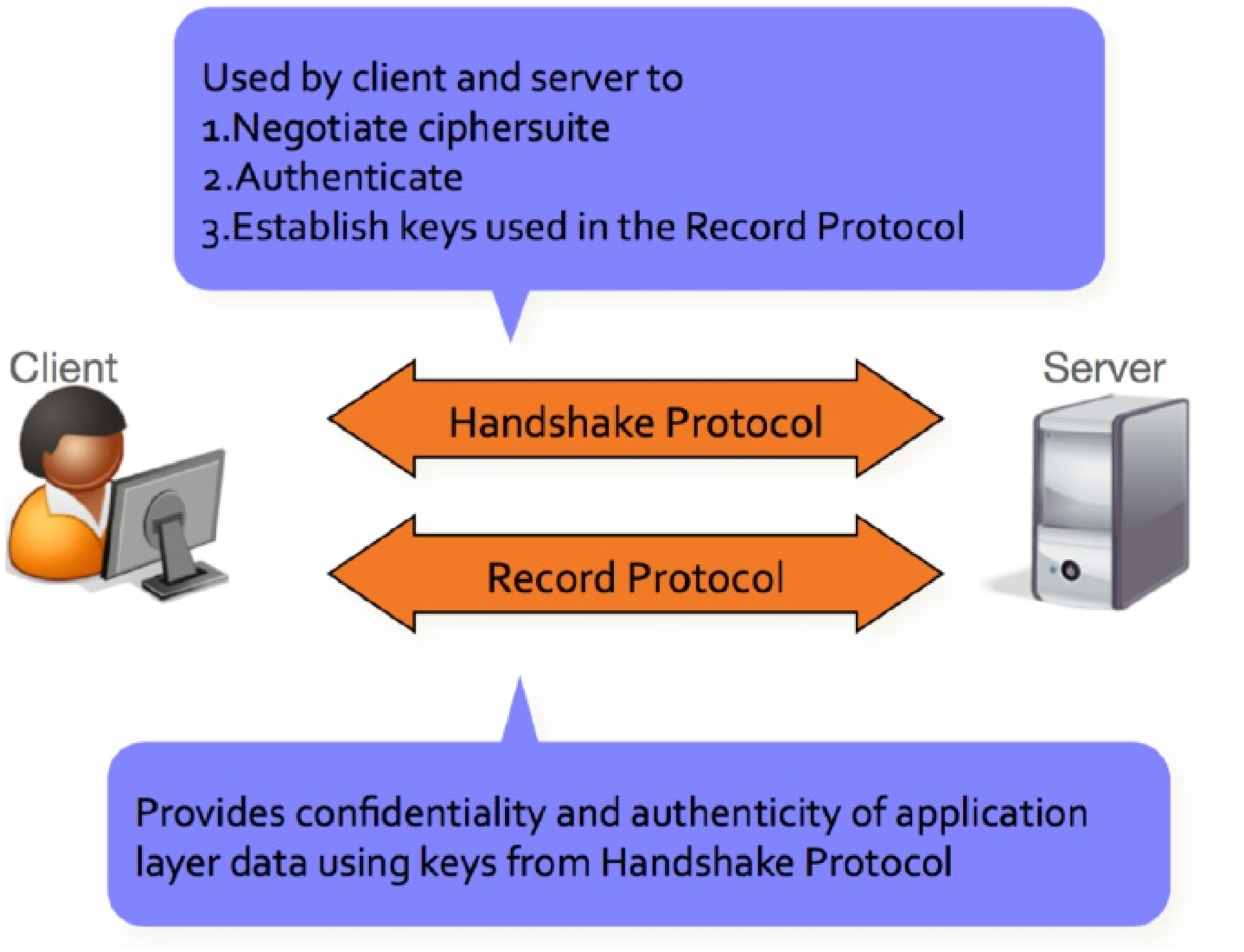
SSL basics

consists of 2 protocols

- Handshake protocol
  - uses public-key cryptography to establish several shared secret keys between client/server
- Record protocol
  - uses secret key established in handshake protocol to protect confidentiality, integrity, Authenticity of data exchange between client/server

TLS


Used by client and server to
1.Negotiate ciphersuite
2.Authenticate
3.Establish keys used in the Record Protocol
Client
Server
Handshake Protocol
Record Protocol
Provides confidentiality and authenticity of application layer data using keys from Handshake Protocol

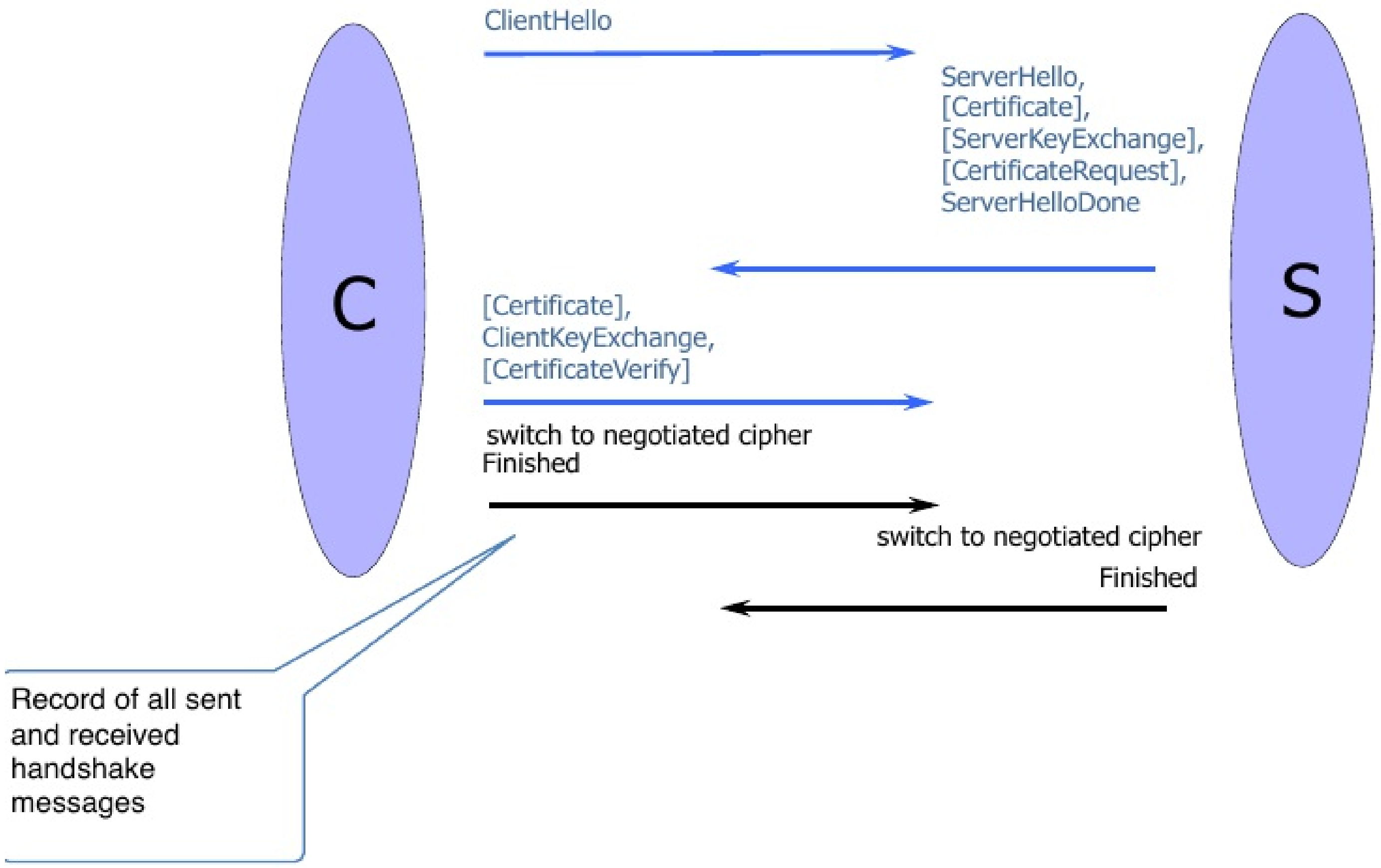

Used by client and server to
1.Negotiate ciphersuite
2.Authenticate
3.Establish keys used in the Record Protocol
Client
Server
Handshake Protocol
Record Protocol
Provides confidentiality and authenticity of application layer data using keys from Handshake Protocol

TLS record protocol

provides:

- data origin authentication, integrity using a MAC
- confidentiality using symmetric encryption algorithm
- anti-replay using sequence numbers protected by MAC
- optional compression
- fragmentation of application layer messages

TLS Handshake Protocol


ClientHello
ServerHello, [Certificate], [ServerKeyExchange], [CertificateRequest], ServerHelloDone
C
S
[Certificate], ClientKeyExchange, [CertificateVerify]
switch to negotiated cipher Finished
switch to negotiated cipher Finished
Record of all sent and received handshake messages

- **ClientHello**
  Client announces in plaintext
  - running protocol version
  - supported cryptographic algorithms
  - fresh, random number
- **ServerHello**
  Server responds in plaintext with
  - highest protocol version supported by client and server
  - strongest cryptographic suite from those offered by client
  - fresh, random number
- **certificate**
  server sends certificate containing RSA or Diffie-Hellman public key
- **ServerKeyExchange**
  optionally server sends temporary key
- **ClientKeyExchange**
  client generates secret key material, sends it to server encrypted with server's public key

Finished messages

- TLS Finished messages enable each side to check that both views of handshake protocol are the same
- comuted as PRF(ms, transcript), transcript=sender's view of all protocol messages sent/received up to this point
- compared by recipient expected value; protocol aborts if mismatch observed
- designed to prevent version rollback/ciphersuite downgrade attacks
  - ineffective if attacker can comute ms during protocol run

Additional features

- TLS Handshake protocol supports renegotiation and session resumption
- renegotiation allows re-keying and change of ciphersuite during session

- initiated by client sending ClientHello or server sending ServerHelloRequest
    - followed by full run of handshake protocol
    - over existing record protocol

sessions and connections

- session resumption allows authentication and shared secrets to be reused across multiple, parallel connections in single session
- session concept:
    - sessions created by handshake protocol
    - state defined by session ID, set of cryptographic parameters negotiated in handshake protocol
    - each session can carry multiple parallel connections
- connection concept:
    - keys for multiple connections derived from single ms created during one run of handshake protocol
    - session resumption handshake protocol runs exchange new nonces
    - nonces combined with existing ms to derive keys for each new connection
    - avoids repeated use of expensive handshake protocol

Attacks

- version rollback attack
- renegotiation attack
- cross-ciphersuite attacks
- FREAK Attack
- LOGJAM Attack
- SLOTH Attack
- TLS Hearbleed