Software Security 101

Erik Poll

Digital Security

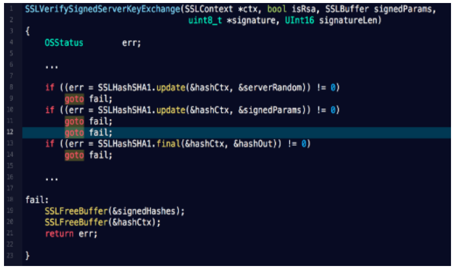
Radboud University Nijmegen

Two ways to create security problems:

1. ‘hack’ the computer

   ie. find a weakness in the software

   eg. exploit a zero-day



```
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;

    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

2. ‘hack’ the user

   incl. social engineering, eg. phishing

Pointing the finger at the user is nearly always victim blaming and a badly designed interface is the real cause

Not just end-users are users, so are sys-admins and developers

So even in 2nd case software is to blame!

Improving security

We do *not* know how to make systems secure
but we do know how to make them (a bit) *more* secure

1st step: Awareness

Realising that security might be an issue

2nd step: Knowledge

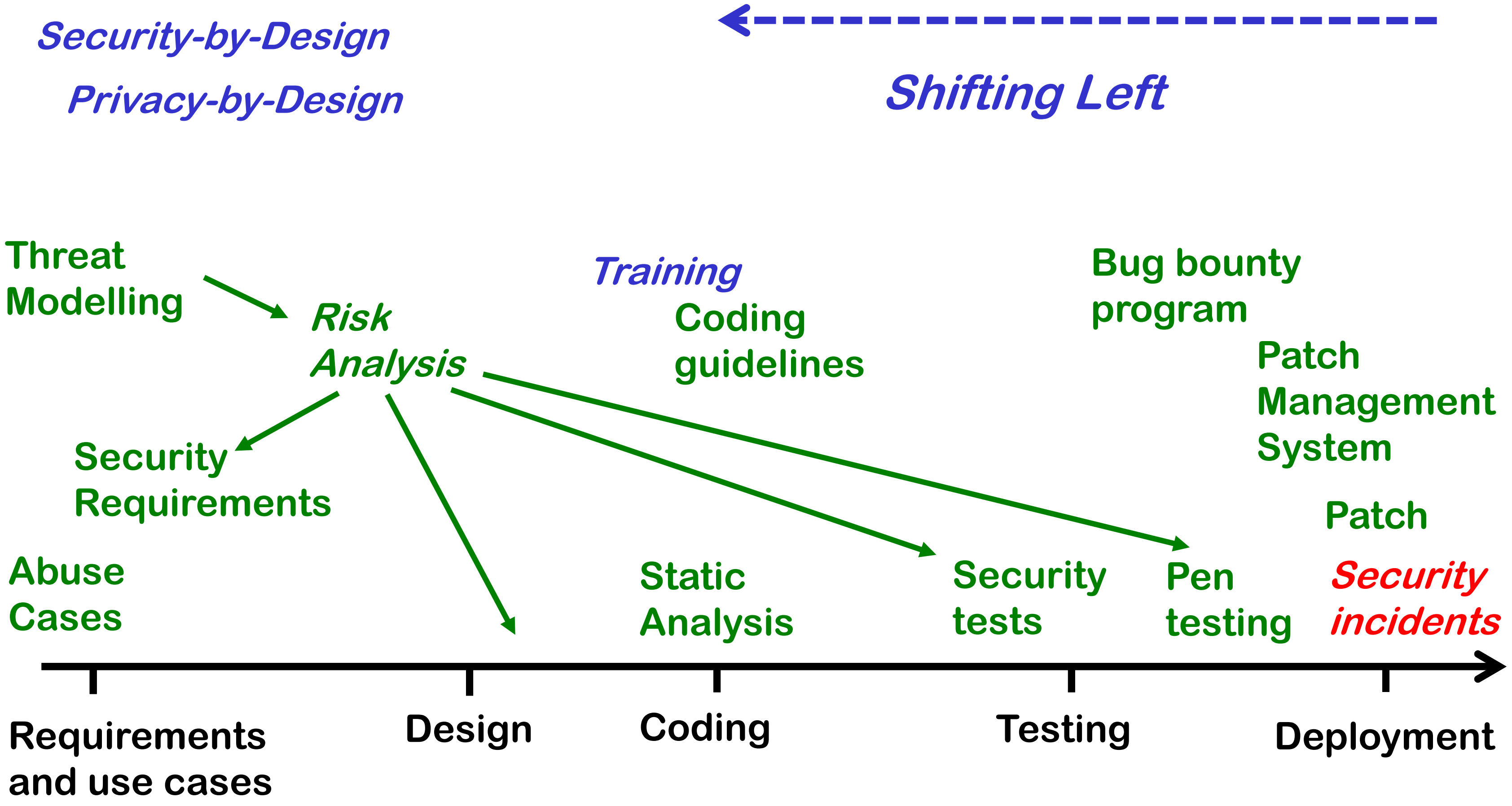
Improving knowledge about security

- LOTS of info available nowadays
- Beware: it depends heavily on platform, programming language, APIs, technology stack, type of application, …

3rd, 4th, … steps: Putting this into practice

Building attention to security into development process

Security in Software Development Lifecycle



Security-by-Design

Privacy-by-Design

Shifting Left

Threat Modelling

Risk Analysis

Security Requirements

Abuse Cases

Training

Coding guidelines

Static Analysis

Security tests

Bug bounty program

Patch Management System

Pen testing

Patch

Security incidents

Requirements and use cases

Design

Coding

Testing

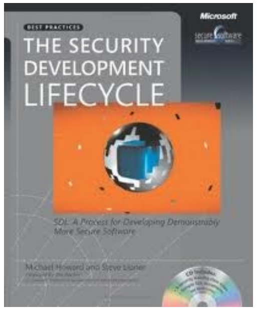
Deployment

DAST, IAST, SAST, RASP

Security people keep inventing new acronyms

- DAST
  - Dynamic Application Security Testing
  - ie. testing
- IAST
  - Interactive Application Security Testing
  - ie. manual testing by eg pen-tester, maybe using DAST tools
- SAST
  - Static Application Security Testing
  - ie. static analysis
- RASP
  - Run-time Application Security Protection
  - ie. monitoring

Plenty of methodologies


Microsoft
BEST PRACTICES
THE SECURITY DEVELOPMENT LIFECYCLE
SDL: A Process for Developing Demonstrably More Secure Software
Michael Howard and Steve Lipner

- **Microsoft SDL**

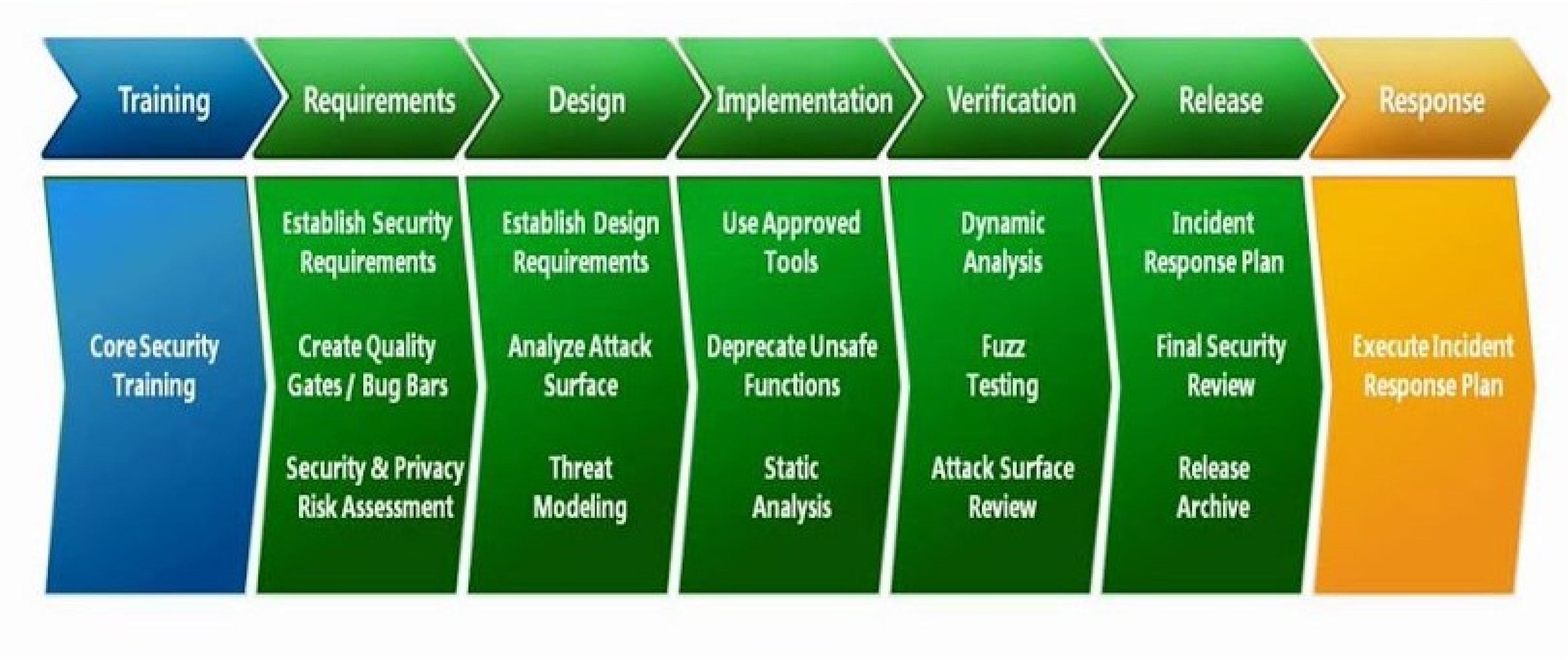
  with extension for Secure DevOps (DevSecOps)
- **BSIMM** **(Building Security In Maturity Model)**
- **Grip op SSD**

  Ongoing initiative by Dutch government organisations

  https://www.cip-overheid.nl/en/category/products/secure-software/
- **…**

These all come with best practices, checklists, methods for assessments, roadmaps for improvements, …

Microsoft SDL



Training
Core Security Training

Requirements
Establish Security Requirements
Create Quality Gates / Bug Bars
Security & Privacy Risk Assessment

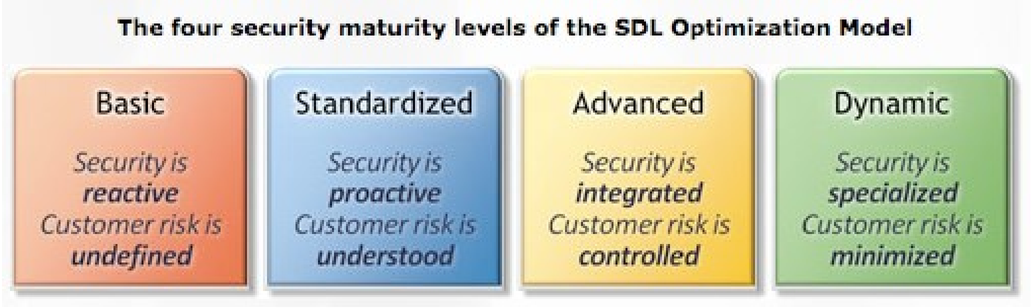
Design
Establish Design Requirements
Analyze Attack Surface
Threat Modeling

Implementation
Use Approved Tools
Deprecate Unsafe Functions
Static Analysis

Verification
Dynamic Analysis
Fuzz Testing
Attack Surface Review

Release
Incident Response Plan
Final Security Review
Release Archive

Response
Execute Incident Response Plan



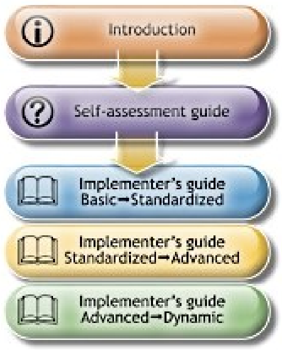
The four security maturity levels of the SDL Optimization Model

Basic
Security is *reactive*
Customer risk is *undefined*

Standardized
Security is *proactive*
Customer risk is *understood*

Advanced
Security is *integrated*
Customer risk is *controlled*

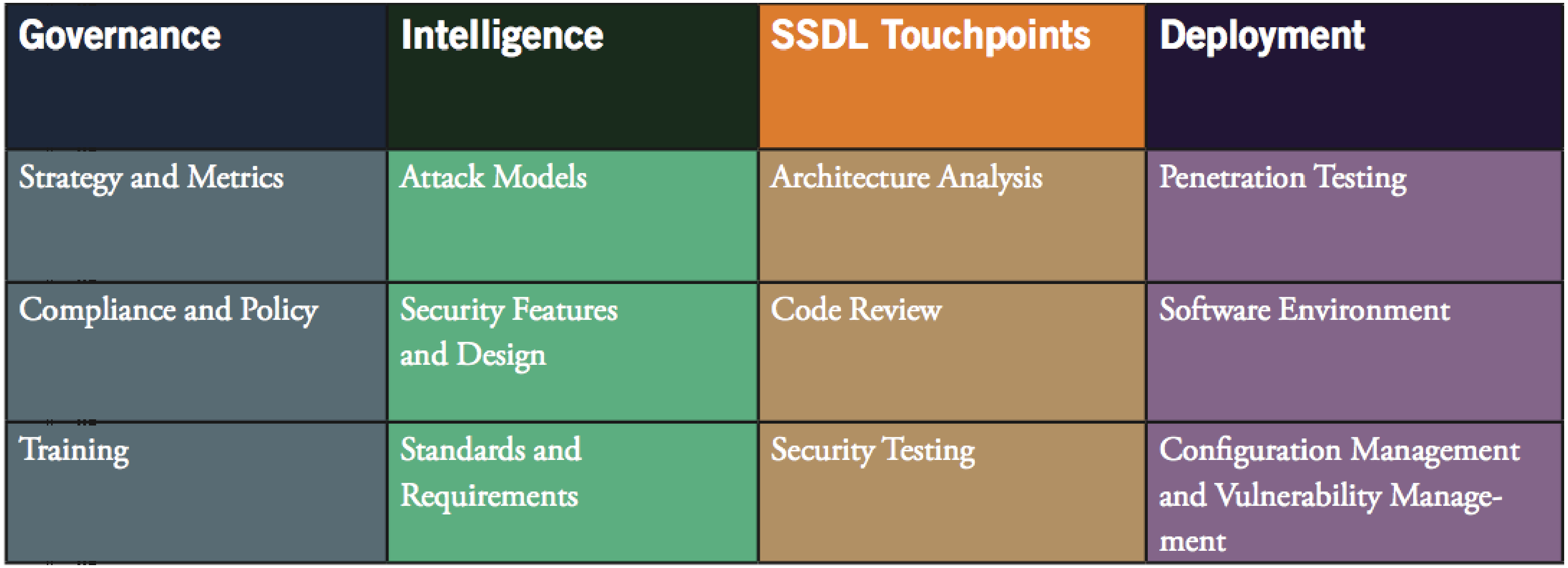
Dynamic
Security is *specialized*
Customer risk is *minimized*



Introduction
Self-assessment guide
Implementer's guide Basic→Standardized
Implementer's guide Standardized→Advanced
Implementer's guide Advanced→Dynamic

BSIMM (Building Security In Maturity Model)

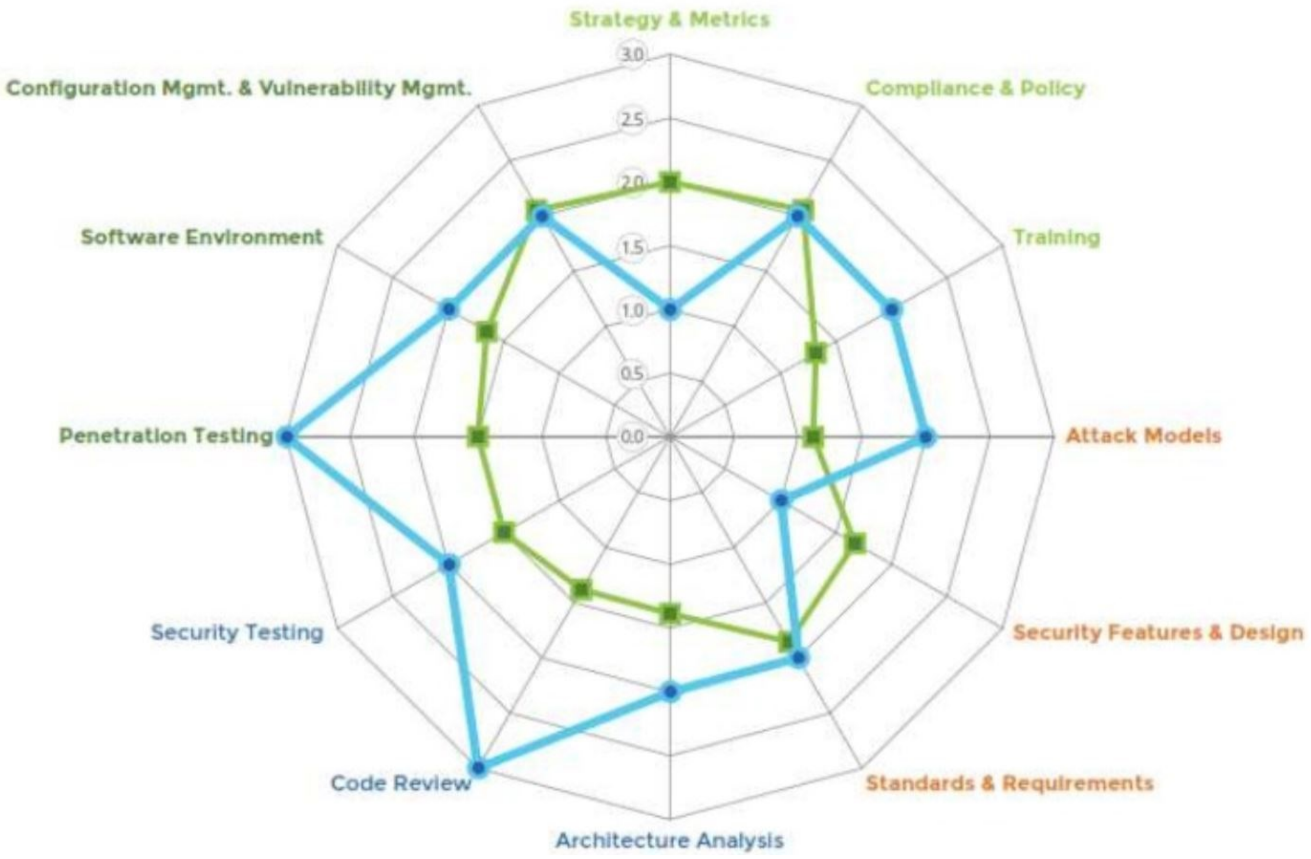
Framework to compare your software security efforts with other organisations



Governance	Intelligence	SSDL Touchpoints	Deployment
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management

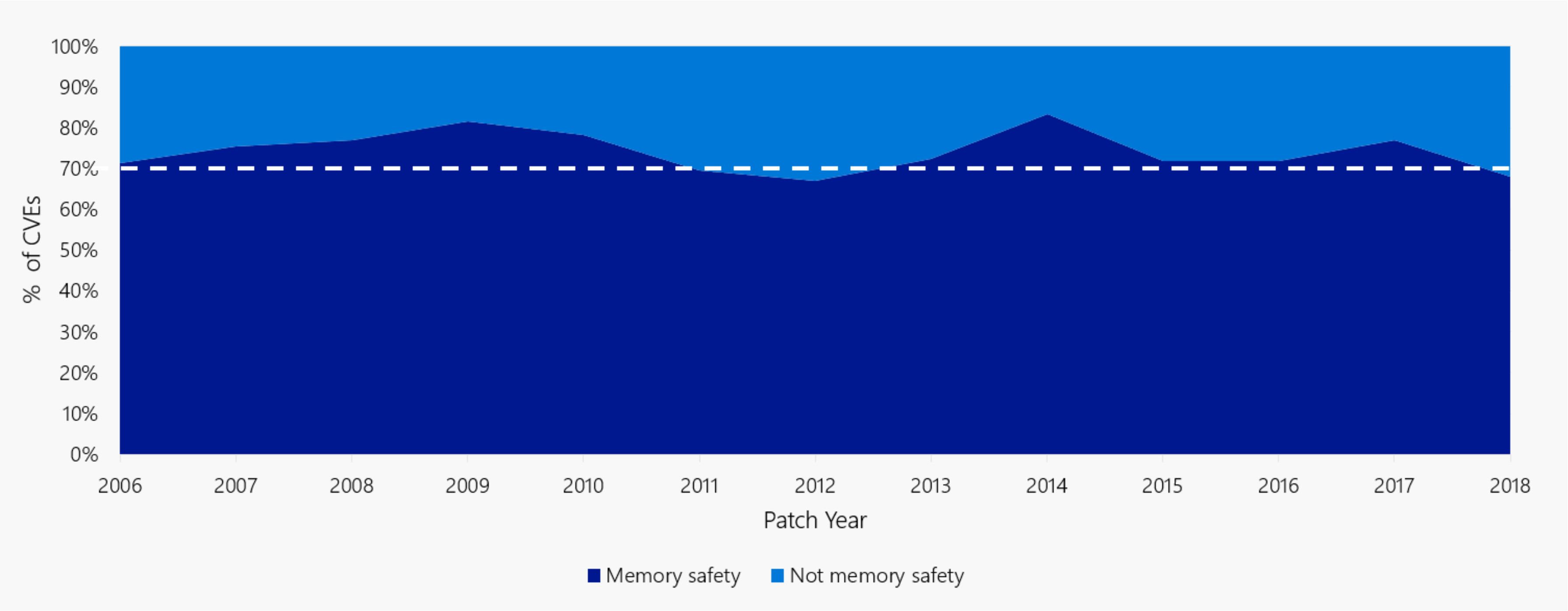
https://www.bsimm.com/framework/

BSIMM: comparing your security maturity



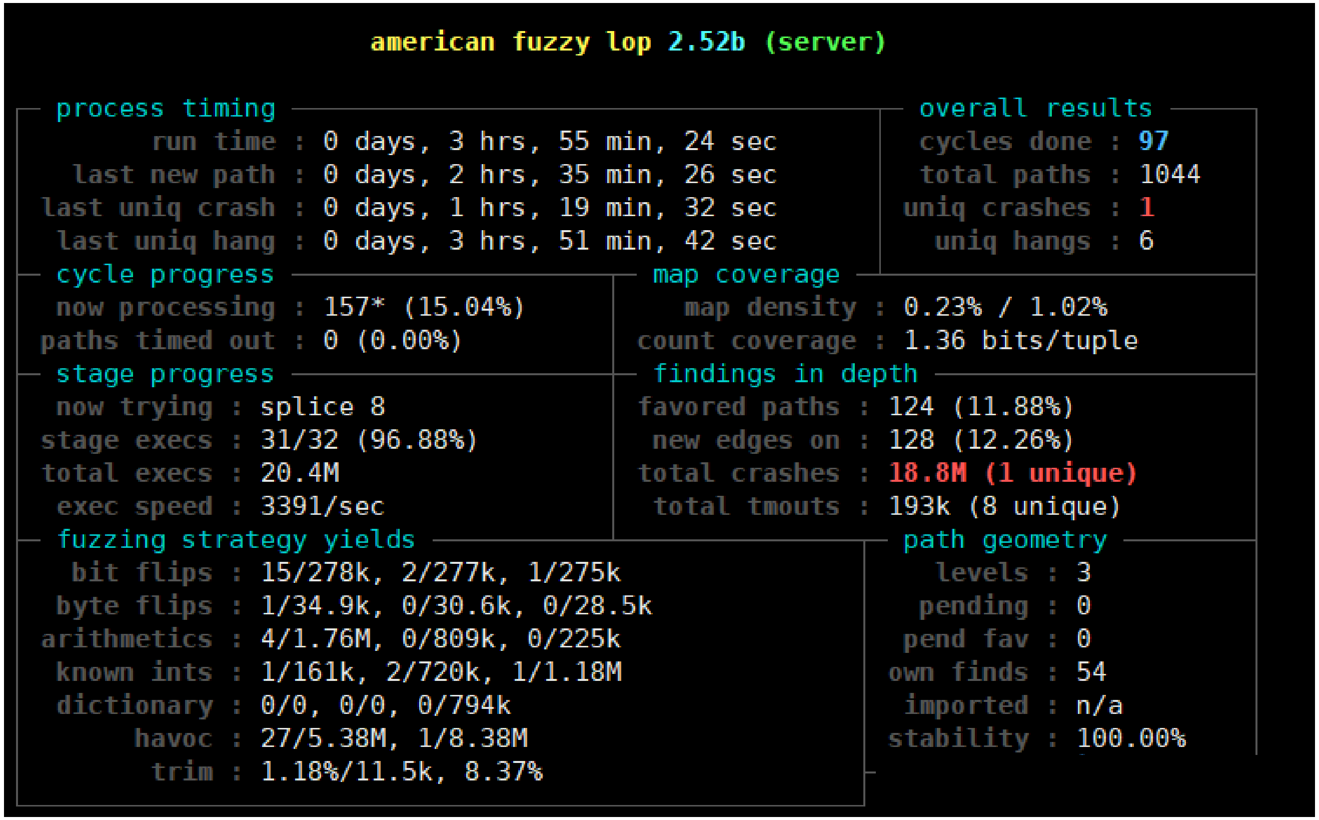
Good practice no 1: use Rust instead of C(++) !

Memory corruption still main source of problems, so using a memory-safe programming language prevents many problems!



Good practice no 2: use a fuzzer!

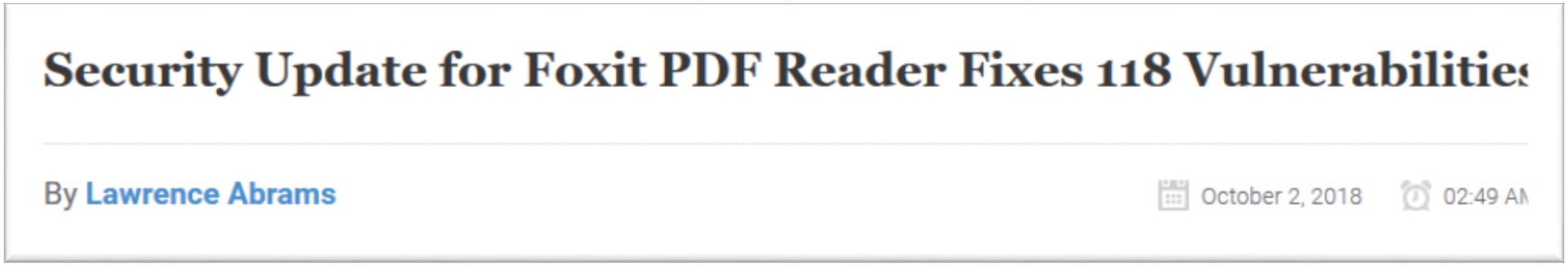
If you have any C(++) code, say in libraries, or unsafe Rust, use a fuzzer! Eg afl++



```
                   american fuzzy lop 2.52b (server)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 3 hrs, 55 min, 24 sec      │  cycles done : 97     │
│   last new path : 0 days, 2 hrs, 35 min, 26 sec      │  total paths : 1044   │
│ last uniq crash : 0 days, 1 hrs, 19 min, 32 sec      │ uniq crashes : 1      │
│  last uniq hang : 0 days, 3 hrs, 51 min, 42 sec      │   uniq hangs : 6      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 157* (15.04%)     │    map density : 0.23% / 1.02%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.36 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 8              │ favored paths : 124 (11.88%)           │
│ stage execs : 31/32 (96.88%)        │  new edges on : 128 (12.26%)           │
│ total execs : 20.4M                 │ total crashes : 18.8M (1 unique)       │
│  exec speed : 3391/sec              │  total tmouts : 193k (8 unique)        │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 15/278k, 2/277k, 1/275k               │    levels : 3          │
│  byte flips : 1/34.9k, 0/30.6k, 0/28.5k             │   pending : 0          │
│ arithmetics : 4/1.76M, 0/809k, 0/225k               │  pend fav : 0          │
│  known ints : 1/161k, 2/720k, 1/1.18M               │ own finds : 54         │
│  dictionary : 0/0, 0/0, 0/794k                      │  imported : n/a        │
│       havoc : 27/5.38M, 1/8.38M                     │ stability : 100.00%    │
│        trim : 1.18%/11.5k, 8.37%                    ├────────────────────────┘
└─────────────────────────────────────────────────────┘
```

The kind of bugs a fuzzer can find



Security Update for Foxit PDF Reader Fixes 118 Vulnerabilities

By Lawrence Abrams

October 2, 2018 02:49 AM

- Root cause: PDF spec is horrendously complex
- These bugs are mainly memory corruption flaws that allow remote code execution
  - so high impact and easy to exploit with email attachments
- *All* PDF viewers suffer from such problems

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=PDF

Audience poll: useful OWASP products

- ***Who here knows the OWASP Top Ten?***

- ***Who here knows the OWASP ASVS?***

**ASVS (Application Security Verification Standard)
takes a more 'constructive' approach than the Top 10
by pointing out things you should do
rather than things that you should *not* do**

The *many* standard security flaws

OWASP Top 10 [2017]

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring

SANS/CWE TOP 25 [2019]

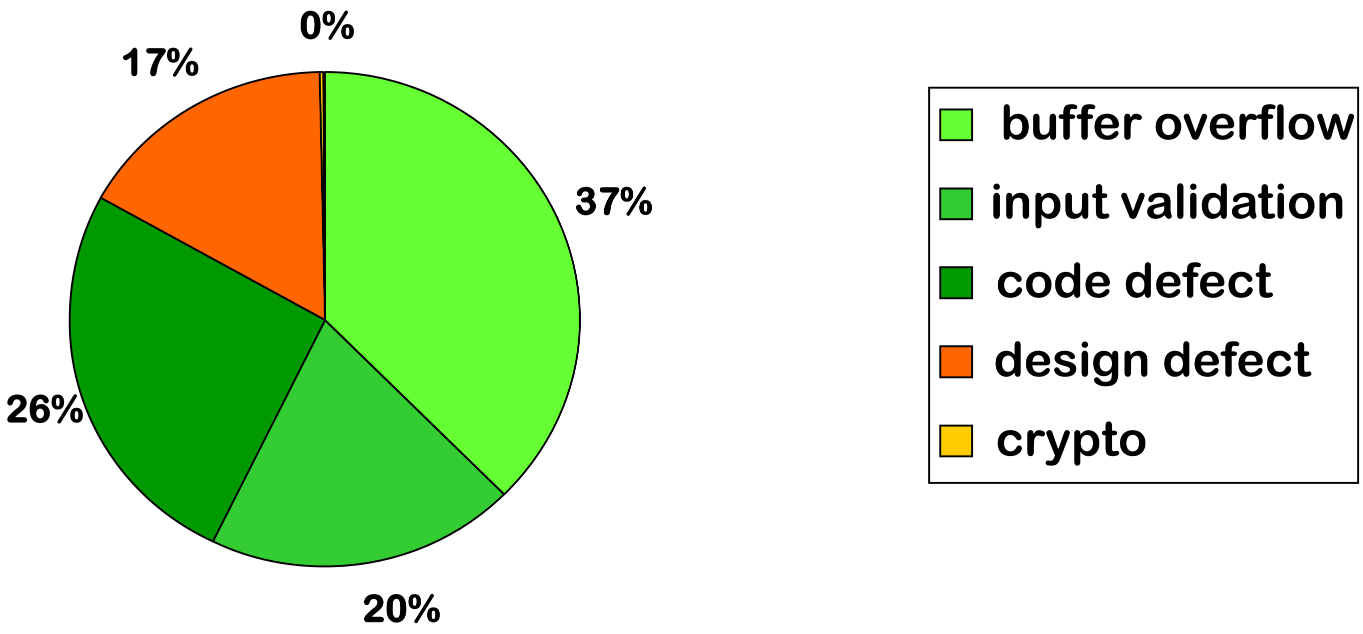
1. Improper Restriction of Operations within the Bounds of a Memory Buffer
2. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3. Improper Input Validation
4. Information exposure
5. Buffer overread
6. SQL Injection
7. Use After Free
8. Integer Overflow
9. CSRF
10. Path Traversal
11. OS Command Injection
12. Out-of-bounds Write
13. Improper Authentication
14. NULL Pointer Dereference
15. Incorrect Permission Assignment
16. Unrestricted Upload of File with Dangerous Type
17. Improper Restriction of XML External Entity
18. Code Injection
19. Use of Hard-coded Credentials
20. Uncontrolled Resource Consumption
21. Missing Release of Resource
22. Untrusted Search Path
23. Deserialization of Untrusted Data
24. Improper Privilege Management
25. Improper Certificate Validation

CWE TOP 924

CWE-14	Compiler Removal of Code to Clear Buffers
CWE-20 ○	Improper Input Validation
CWE-22 ○	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-23 ○	Relative Path Traversal
CWE-24 ○	Path Traversal: '../filedir'
CWE-25 ○	Path Traversal: '/../filedir'
CWE-26 ○	Path Traversal: '/dir/../filename'
CWE-27 ○	Path Traversal: 'dir/../../filename'
CWE-28 ○	Path Traversal: '..\filedir'
CWE-29 ○	Path Traversal: '\..\filename'
CWE-30 ○	Path Traversal: '\dir\..\filename'
CWE-31 ○	Path Traversal: 'dir\..\..\filename'
CWE-32 ○	Path Traversal: '...' (Triple Dot)
CWE-33 ○	Path Traversal: '....' (Multiple Dot)
CWE-34 ○	Path Traversal: '....//'
CWE-35 ○	Path Traversal: '.../...//'
CWE-36 ○	Absolute Path Traversal
CWE-37 ○	Path Traversal: '/absolute/pathname/here'
CWE-38 ○	Path Traversal: '\absolute\pathname\here'
CWE-39 ○	Path Traversal: 'C:dirname'
CWE-40 ○	Path Traversal: '\\UNC\share\name\' (Windows UNC Share)
CWE-41 ○	Improper Resolution of Path Equivalence
CWE-51 ○	Path Equivalence: '/multiple//internal/slash'
CWE-55 ○	Path Equivalence: '/./' (Single Dot Directory)
CWE-57 ○	Path Equivalence: 'fakedir/../realdir/filename'
CWE-59 ○	Improper Link Resolution Before File Access ('Link Following')
CWE-61	UNIX Symbolic Link (Symlink) Following
CWE-62	UNIX Hard Link
CWE-73	External Control of File Name or Path
CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component('Injection')
CWE-75	Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)
CWE-76	Improper Neutralization of Equivalent Special Elements
CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS CommandInjection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-88	Argument Injection or Modification
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
CWE-91	XML Injection (aka Blind XPath Injection)
CWE-93	Improper Neutralization of CRLF Sequences ('CRLF Injection')
CWE-94	Improper Control of Generation of Code ('Code Injection')
CWE-95	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')
CWE-96	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')
CWE-97	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page
CWE-99	Improper Control of Resource Identifiers ('Resource Injection')
CWE-114	Process Control
CWE-116	Improper Encoding or Escaping of Output
CWE-117	Improper Output Neutralization for Logs
CWE-123	Write-what-where Condition
CWE-134	Use of Externally-Controlled Format String
CWE-135	Incorrect Calculation of Multi-Byte String Length
CWE-138	Improper Neutralization of Special Elements
CWE-140	Improper Neutralization of Delimiters
CWE-141	Improper Neutralization of Parameter/Argument Delimiters
CWE-142	Improper Neutralization of Value Delimiters
CWE-143	Improper Neutralization of Record Delimiters
CWE-144	Improper Neutralization of Line Delimiters
CWE-145	Improper Neutralization of Section Delimiters
CWE-146	Improper Neutralization of Expression/Command Delimiters
CWE-147	Improper Neutralization of Input Terminators
CWE-148	Improper Neutralization of Input Leaders
CWE-149	Improper Neutralization of Quoting Syntax
CWE-150	Improper Neutralization of Escape, Meta, or Control Sequences
CWE-151	Improper Neutralization of Comment Delimiters
CWE-152	Improper Neutralization of Macro Symbols
CWE-153	Improper Neutralization of Substitution Characters
CWE-154	Improper Neutralization of Variable Name Delimiters
CWE-155	Improper Neutralization of Wildcards or Matching Symbols
CWE-156	Improper Neutralization of Whitespace
CWE-157	Failure to Sanitize Paired Delimiters
CWE-158	Improper Neutralization of Null Byte or NUL Character
CWE-159	Failure to Sanitize Special Element
CWE-160	Improper Neutralization of Leading Special Elements
CWE-161	Improper Neutralization of Multiple Leading Special Elements
CWE-162	Improper Neutralization of Trailing Special Elements
CWE-163	Improper Neutralization of Multiple Trailing Special Elements
CWE-164	Improper Neutralization of Internal Special Elements
CWE-165	Improper Neutralization of Multiple Internal Special Elements
CWE-166	Improper Handling of Missing Special Element
CWE-167	Improper Handling of Additional Special Element
CWE-168	Improper Handling of Inconsistent Special Elements
CWE-172	Encoding Error
CWE-173	Improper Handling of Alternate Encoding
CWE-174	Double Decoding of the Same Data
CWE-175	Improper Handling of Mixed Encoding
CWE-176	Improper Handling of Unicode Encoding
CWE-177	Improper Handling of URL Encoding (Hex Encoding)
CWE-178	Improper Handling of Case Sensitivity
CWE-179	Incorrect Behavior Order: Early Validation
CWE-180	Incorrect Behavior Order: Validate Before Canonicalize
CWE-181	Incorrect Behavior Order: Validate Before Filter
CWE-182	Collapse of Data into Unsafe Value
CWE-184 ○	Incomplete Blacklist
CWE-185	Incorrect Regular Expression
CWE-186	Overly Restrictive Regular Expression
CWE-187	Partial Comparison
CWE-188 ○	Reliance on Data/Memory Layout
CWE-200	Information Exposure
CWE-201	Information Exposure Through Sent Data
CWE-203	Information Exposure Through Discrepancy
CWE-204	Response Discrepancy Information Exposure
CWE-209	Information Exposure Through an Error Message
CWE-210	Information Exposure Through Self-generated Error Message
CWE-211	Information Exposure Through Externally-generated Error Message
CWE-212	Improper Cross-boundary Removal of Sensitive Data
CWE-215	Information Exposure Through Debug Information
CWE-216	Containment Errors (Container Errors)
CWE-227 ○	Improper Fulfillment of API Contract ('API Abuse')
CWE-241	Improper Handling of Unexpected Data Type
CWE-252	Unchecked Return Value
CWE-253	Incorrect Check of Function Return Value
CWE-273	Improper Check for Dropped Privileges
CWE-311	Missing Encryption of Sensitive Data
CWE-319	Cleartext Transmission of Sensitive Information
CWE-354	Improper Validation of Integrity Check Value
CWE-364 ◄	Signal Handler Race Condition
CWE-365 ◄	Race Condition in Switch
CWE-374	Passing Mutable Objects to an Untrusted Method
CWE-375	Returning a Mutable Object to an Untrusted Caller
CWE-378	Creation of Temporary File With Insecure Permissions
CWE-379	Creation of Temporary File in Directory with Incorrect Permissions
CWE-390	Detection of Error Condition Without Action
CWE-391	Unchecked Error Condition
CWE-394	Unexpected Status Code or Return Value
CWE-405 ◄	Asymmetric Resource Consumption (Amplification)
CWE-406	Insufficient Control of Network Message Volume (Network Amplification)
CWE-407 ○	Algorithmic Complexity
CWE-408 ◄	Incorrect Behavior Order: Early Amplification
CWE-409	Improper Handling of Highly Compressed Data (Data Amplification)
CWE-410	Insufficient Resource Pool
CWE-412 ◄	Unrestricted Externally Accessible Lock
CWE-413 ◄	Improper Resource Locking
CWE-414 ◄	Missing Lock Check
CWE-430	Deployment of Wrong Handler
CWE-431	Missing Handler
CWE-432 ◄	Dangerous Signal Handler not Disabled During Sensitive Operations
CWE-447 ○	Unimplemented or Unsupported Feature in UI
CWE-453	Insecure Default Variable Initialization
CWE-454	External Initialization of Trusted Variables or Data Stores
CWE-455	Non-exit on Failed Initialization
CWE-456	Missing Initialization of a Variable
CWE-460	Improper Cleanup on Thrown Exception
CWE-462	Duplicate Key in Associative List (Alist)
CWE-463	Deletion of Data Structure Sentinel
CWE-464	Addition of Data Structure Sentinel
CWE-470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
CWE-472	External Control of Assumed-Immutable Web Parameter
CWE-474 ○	Use of Function with Inconsistent Implementations
CWE-479 ◄	Signal Handler Use of a Non-reentrant Function
CWE-488 ◄	Exposure of Data Element to Wrong Session
CWE-489 ○	Leftover Debug Code
CWE-493 ○	Critical Public Variable Without Final Modifier
CWE-494	Download of Code Without Integrity Check
CWE-496	Public Data Assigned to Private Array-Typed Field
CWE-497	Exposure of System Data to an Unauthorized Control Sphere
CWE-498 ○	Cloneable Class Containing Sensitive Information
CWE-500 ○	Public Static Field Not Marked Final
CWE-502 ○	Deserialization of Untrusted Data
CWE-506 ○	Embedded Malicious Code
CWE-507 ○	Trojan Horse
CWE-508	Non-Replicating Malicious Code
CWE-509 ○	Replicating Malicious Code (Virus or Worm)
CWE-510	Trapdoor
CWE-511 ○	Logic/Time Bomb
CWE-512 ○	Spyware
CWE-524 ○	Information Exposure Through Caching
CWE-526	Information Exposure Through Environmental Variables
CWE-538	File and Directory Information Exposure
CWE-539 ○	Information Exposure Through Persistent Cookies
CWE-543 ◄	Use of Singleton Pattern Without Synchronization in a Multithreaded Context
CWE-544	Missing Standardized Error Handling Mechanism
CWE-546 ○	Suspicious Comment
CWE-548 ○	Information Exposure Through Directory Listing
CWE-584	Return Inside Finally Block
CWE-587	Assignment of a Fixed Address to a Pointer
CWE-591	Sensitive Data Storage in Improperly Locked Memory
CWE-595	Comparison of Object References Instead of Object Contents
CWE-598	Information Exposure Through Query Strings in GET Request
CWE-605	Multiple Binds to the Same Port
CWE-622 ○	Improper Validation of Function Hook Arguments
CWE-636 ○	Not Failing Securely ('Failing Open')
CWE-637 ○	Unnecessary Complexity in Protection Mechanism (Not Using 'Economy of Mechanism')
CWE-638	Not Using Complete Mediation
CWE-641	Improper Restriction of Names for Files and Other Resources
CWE-643	Improper Neutralization of Data within XPath Expressions ('XPath Injection')
CWE-652	Improper Neutralization of Data within XQuery Expressions ('XQuery Injection')
CWE-663 ◄	Use of a Non-reentrant Function in a Concurrent Context
CWE-664	Improper Control of a Resource Through its Lifetime
CWE-666 ○	Operation on Resource in Wrong Phase of Lifetime
CWE-674 ○	Uncontrolled Recursion
CWE-688	Function Call With Incorrect Variable or Reference as Argument
CWE-694	Use of Multiple Resources with Duplicate Identifier
CWE-754	Improper Check for Unusual or Exceptional Conditions
CWE-759	Use of a One-Way Hash without a Salt

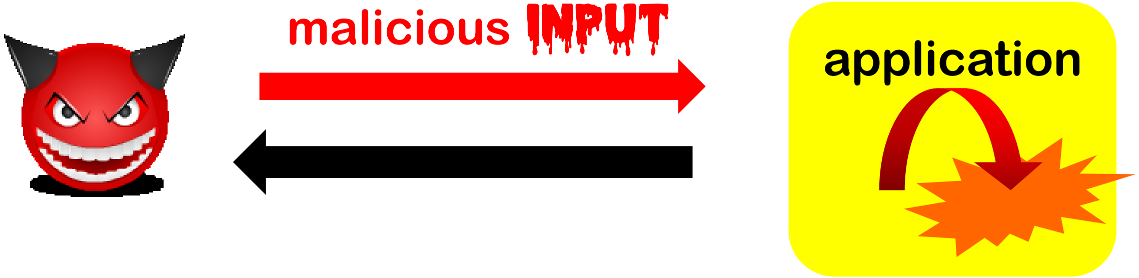
Design vs Implementation flaws

Useful, high level classification



Flaws found in Microsoft's first security bug fix month

The *one* standard security flaw: INPUT handling

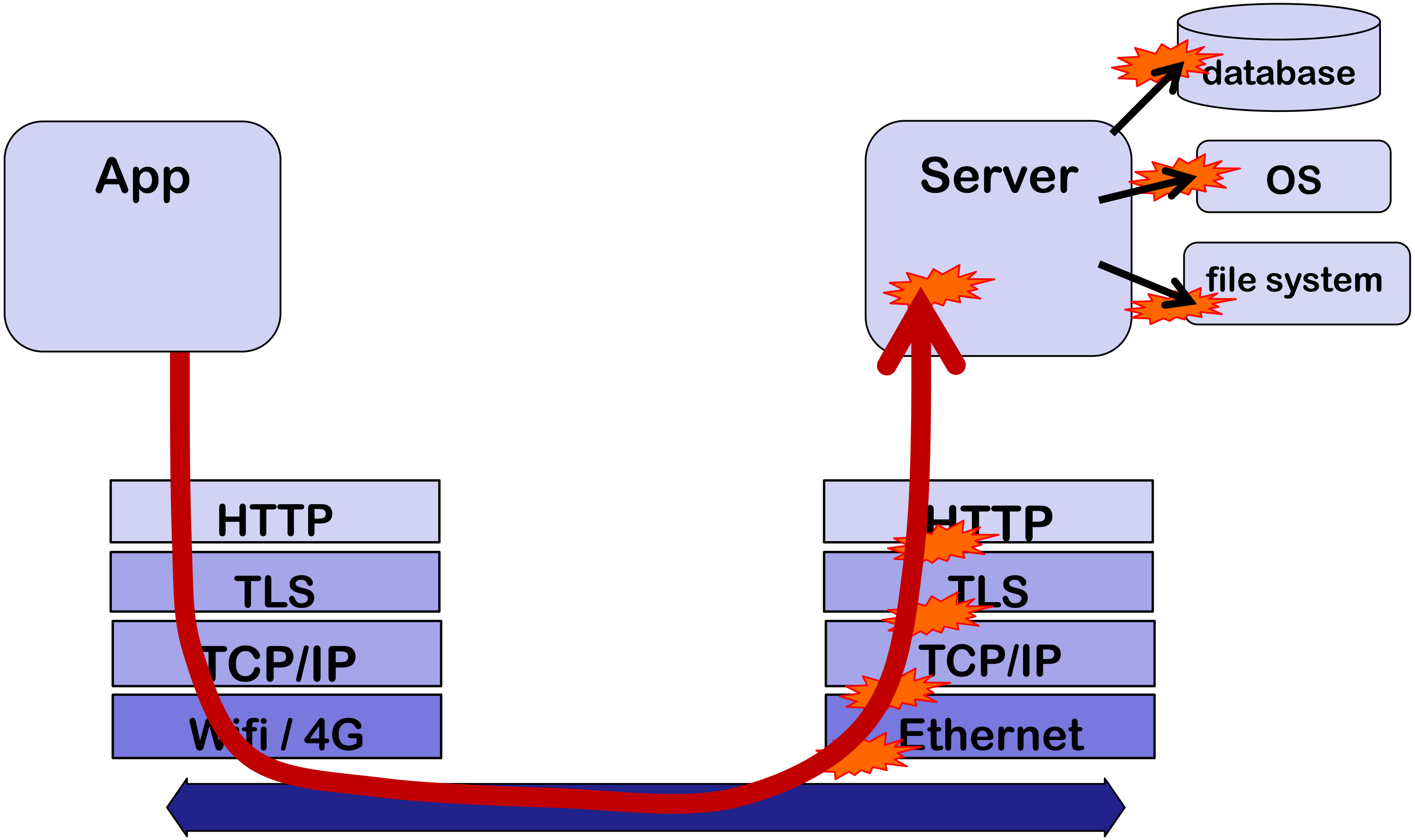


Garbage In, Garbage Out

quickly becomes

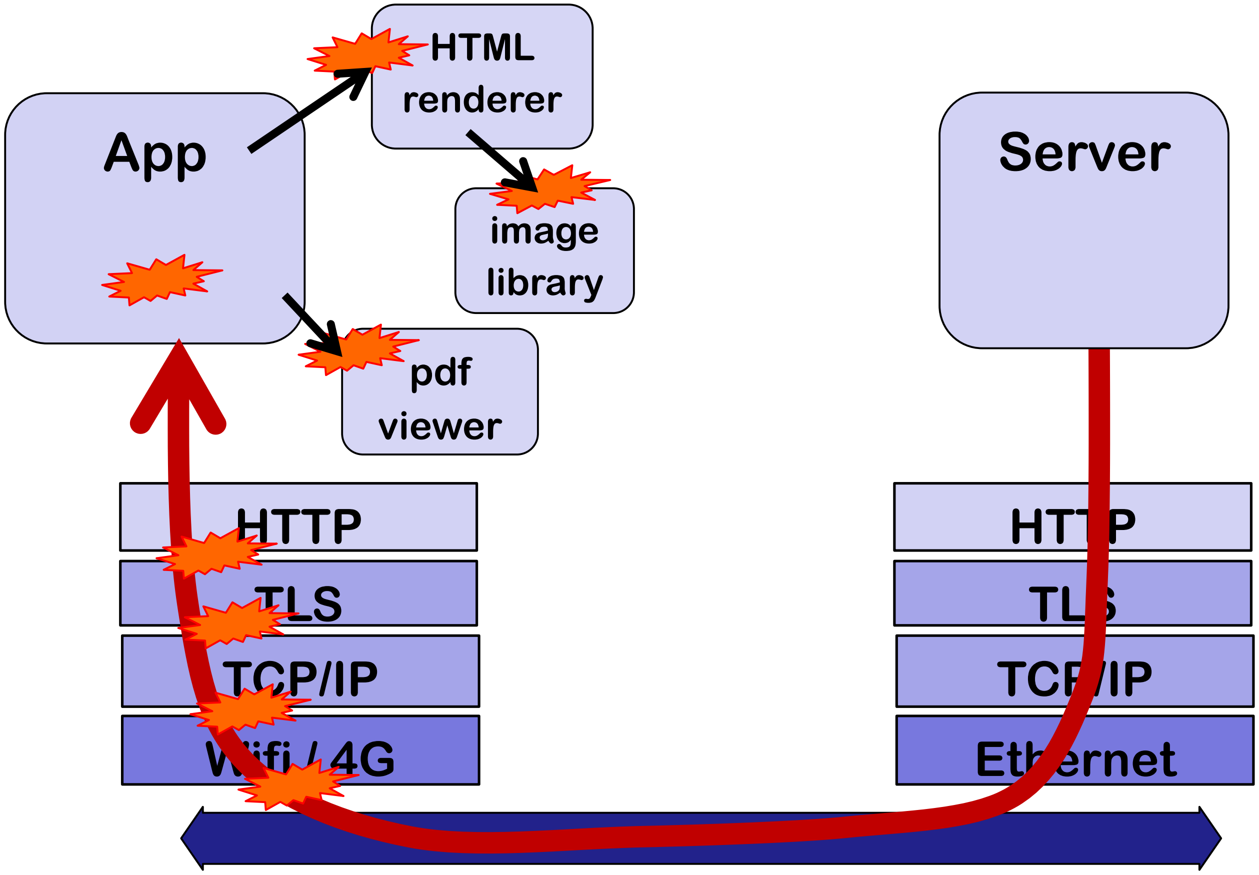
Malicious Garbage In, *Security Incident* Out

Attack surface



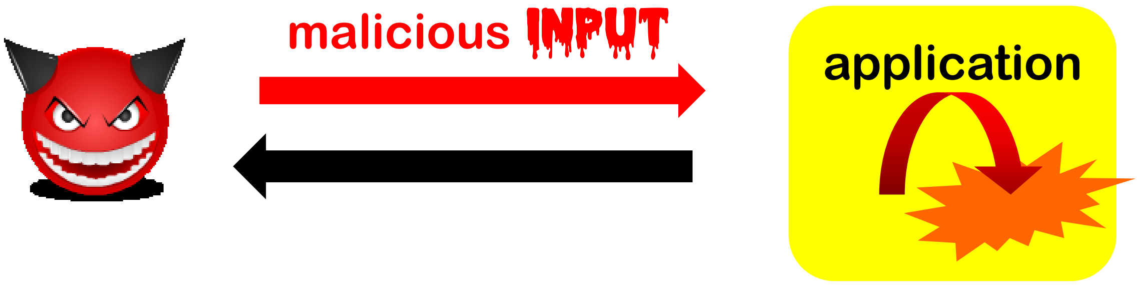
Data is parsed/decoded/interpreted/… as it moves up the technology stack

Attack surface



Data is parsed/decoded/interpreted/... as it moves up the technology stack

Most input problems: PARSING problems



Input only become dangerous when you start PARSING it.

- Your parser could buggy
  - esp. if it is written in C(++)
  - esp. if the input language/format is complex
- You could be parsing & then processing user input (= attacker input!) in ways that is dangerous
  - eg parsing user input as HTML, giving rise to XSS

Root cause: complexity & expressivity in formats/languages

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 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: pass the hash

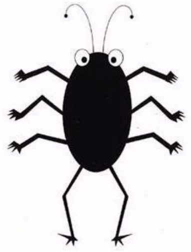
This can be exploited by SMB relay attacks

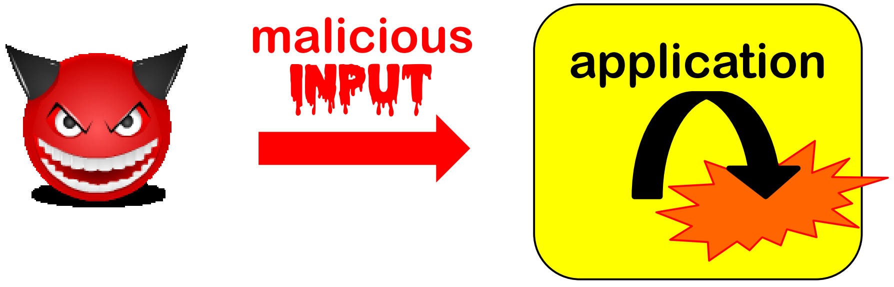
- CVE-2000-0834 in Windows telnet
- CVE-2008-4037 in Windows XP/Server/Vista
- CVE-2016-5166 in Chromium
- CVE-2017-3085 & CVE-2016-4271 in Adobe Flash
- ZDI-16-395 in Foxit PDF viewer

Two types of INPUT problems

1. Processing Flaws

a bug !

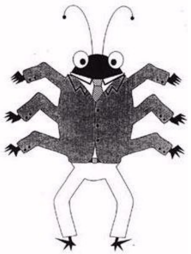


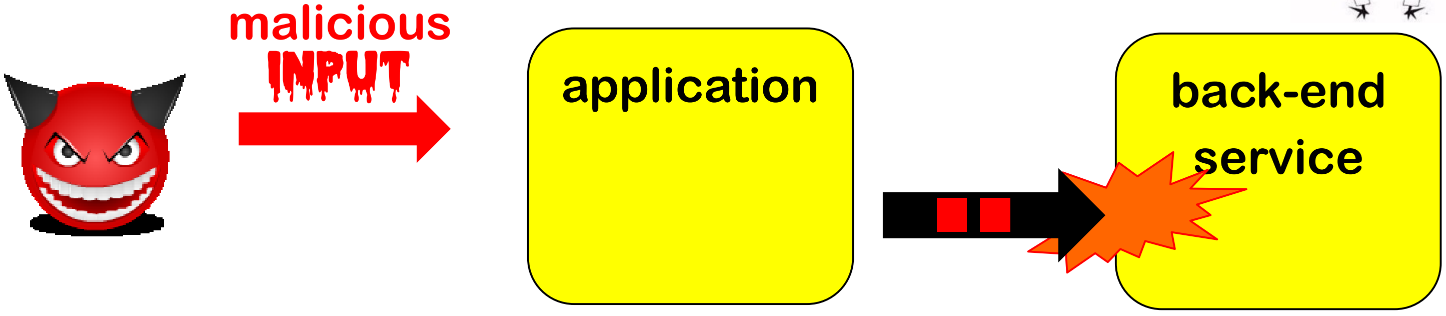


eg buffer overflow in PDF viewer

2. Forwarding/Injection Flaws

(abuse of)
a feature !





eg SQLi, XSS, Word macros, …

Two types of INPUT problems

1. Buggy parsing & processing
   - Bug in processing input causes application to go of the rails
   - Classic example: buffer overflow in a PDF viewer, leading to remote code execution

   This is *unintended* behaviour, introduced by *mistake*

2. Flawed forwarding (aka injection attacks)
   - Input is forwarded to *back-end* service/system/API, to cause damage there
   - Classic examples: SQL injection, path traversal, XSS, Word macros

   This is *intended* behaviour of the back-end, introduced *deliberately*, but *exposed by mistake* by the front-end

Remedies? sanitisation ≠ validation

Often confused but are very different:

- Sanitisation aka escaping aka encoding:
  'fixing' data to make it 'harmless'

  Eg replacing `<` with `&lt;` to prevent XSS

  or `'` with `\'` to prevent SQL injection

  Need to sanitise comes from weakness in back-end interface

  Need is external to the use case, but depends on technologies/APIs used

- Validation: rejecting data because it is invalid

  Eg rejecting `31/11/2021` as a valid date

  Need to reject invalid data stems from the use case/application

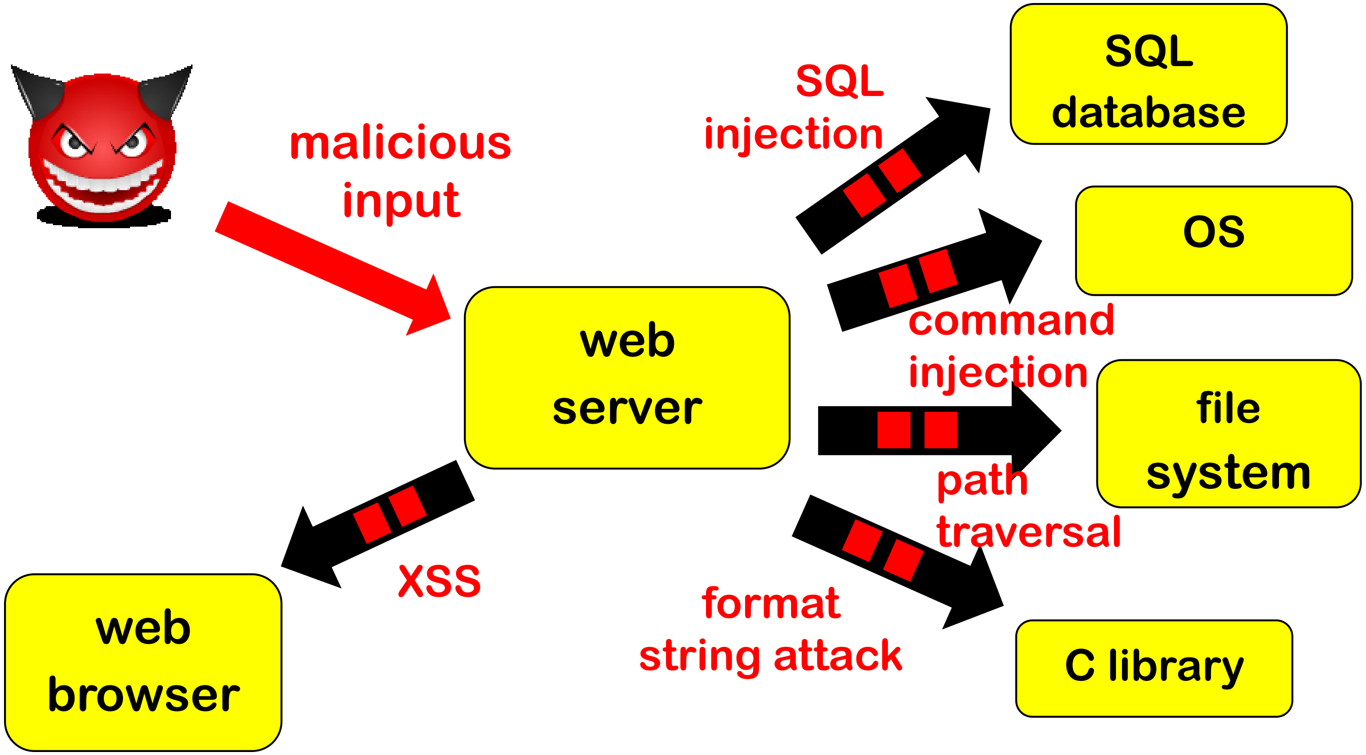
  Validation of input is needed irrespective of whether backend APIs are immune to injection attacks

  Need is inherent to the use case

Input *validation* & output *sanitisation*

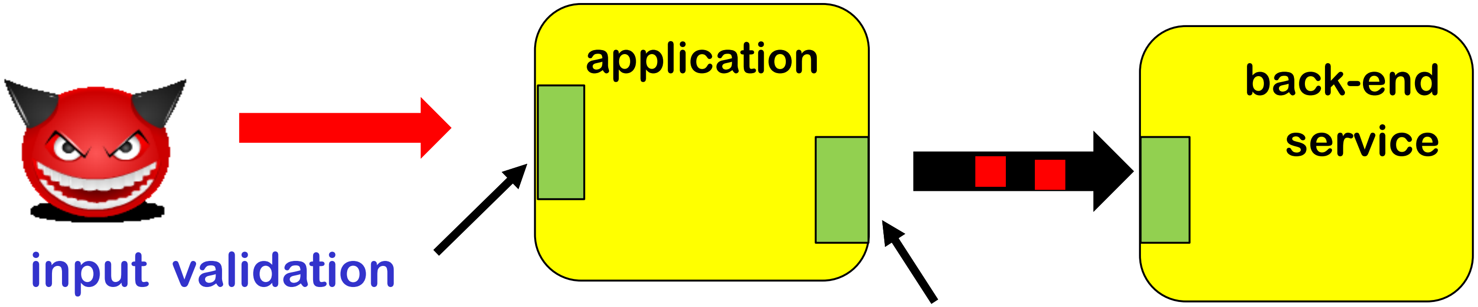
- Input validation is good approach
- Input sanitisation (aka escaping aka encoding) less so
  - Because at the point of input, the context in which input is used (eg. in SQL query or HTML or file name …) is unclear, and different contexts require different sanitisations.

More back-ends, more languages, more problems



Input *validation* & output *sanitisation*

- Input validation is good approach
- Input sanitisation less so
  - Because at the point of input, the context in which input is used (eg. in SQL query or HTML or file name …) is unclear, and different contexts require different sanitisations.
- *Output* sanitisation does makes sense, because there context is known



application

back-end service

input validation
rejecting invalid input

output sanitisation
aka 'fixing' output to make it harmless for this particular back-end

Strings

`String` is a useful datatype because it is so versatile

Eg. a string can be

- a username
- a date
- an email address
- a URL
- a snippet of HTML
- a snippet of SQL
- path name
- directory name
- ...

Strings in web-applications

Here a string can be

- a URL
- a URL that is pointing to a ‘trusted’ domain from which it is safe to download & excute JavaScript
- a URL for which parameters have been HTML-encoded so they do not do contain JavaScript
- a snippet of HTML
- a snippet of HTML that we know does not contain JavaScript (eg because it has been HTML-encoded)
- a ‘trusted’ snippet of HTML that may contain JavaScript but is safe to execute (because it comes from a trusted source)
- text that is JavaScript-literal-encoded, so that is safe to use as JavaScript string parameter
- text that has been first HTML-encoded and then JS-string-literal-encoded
- text that has been first JS-string-literal-encoded and then HTML-encoded
- ….. AARGH

Anti-pattern: STRINGS ⚠

Strings are dangerous in programs because you have no clue

- if a string is meant to be a username, email address, file name, path name, URL, shell command, bit of SQL, HTML, ..
- if it is has been validated, sanatised/escaped, URL/HTML/JS-string-literal/based64/…-encoded, …
- if it is or contains user-controlled input that makes it dangerous to feed it to some of the many back-ends

Better solution: use different TYPES for data of different kinds and of different trust levels

Eg. Google Trusted Types API that replaces the string-based DOM API

[Wang et al., If It's Not Secure, It Should Not Compile:
Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening, ICSE'21]

Software security: Do's

1. Know the typical problems in your technology stack
2. Check best practices of SDL, BSIMM, … that work for you
3. Use memory safe languages
4. Use fuzzers
5. Be careful with parsing
6. Validate inputs & sanitise outputs

   better still, have 'safe' interfaces with back-ends that do not require sanitisation to be used safely

7. Don't use strings, but types that distinguish languages & trust levels

Steps 3-7 catch low-hanging fruit,
not the 'deeper', application-specific bugs … ☹

Thanks for your attention!