Parsing and Network Security

- Postel's law
- Parsing

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Remotely Exploitable Vulnerabilities

- Writing safe code in unsafe languages is difficult
  - Easy to rationalise each individual bug – “How could anyone write that?”

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

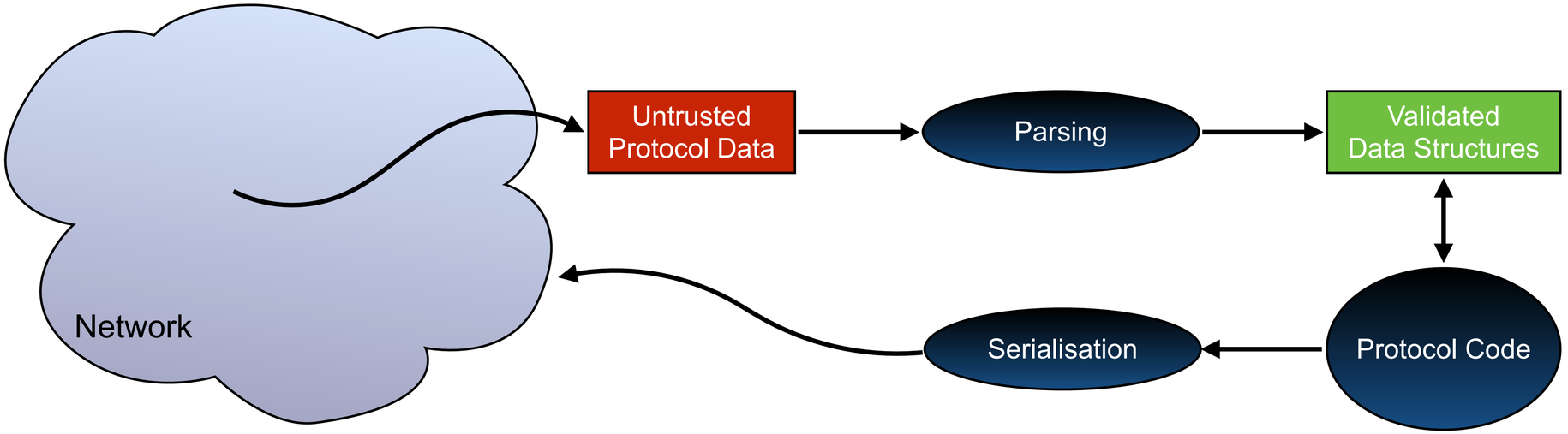
https://dwheeler.com/essays/apple-goto-fail.html

  - But, people will continue to make mistakes
- Remotely exploitable vulnerabilities threaten any networked system

- **Are there particular classes of bug that cause such vulnerabilities?**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Parsing Untrusted Input



- The input parser is critical to security of a networked system
  - Takes untrusted input from the network
  - Generates validated, strongly typed, data structures that are processed by the rest of the code
- How can we ensure parsers are safe?

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

The Robustness Principle (Postel's Law)

> At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability:
>
> "Be liberal in what you accept, and conservative in what you send"
>
> Software should be written to deal with every conceivable error, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue. In general, it is best to assume that the network is filled with malevolent entities that will send in packets designed to have the worst possible effect. This assumption will lead to suitable protective design, although the most serious problems in the Internet have been caused by un-envisaged mechanisms triggered by low-probability events; mere human malice would never have taken so devious a course!

RFC1122

Traditional guidance when writing networked systems is expressed in Postel's law

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

The Robustness Principle (Postel's Law)

"Be liberal in what you accept, and conservative in what you send"

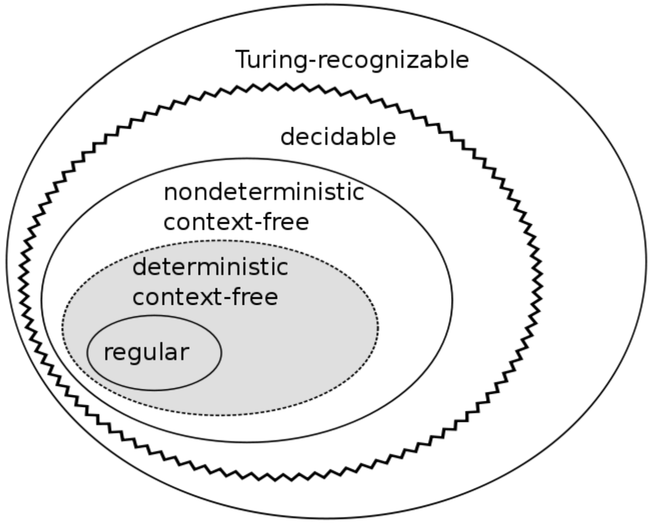
"Postel lived on a network with all his friends.
We live on a network with all our enemies.
Postel was wrong for todays internet."
— *Poul-Henning Kamp*

See also: https://datatracker.ietf.org/doc/draft-iab-protocol-maintenance/

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Define Legal Inputs and Failure Responses

- The input parser takes untrusted input from the network, generates validated, strongly typed, data structures that are processed by the rest of the code
- A parser is an rule-driven automaton
  - It parses the input according to some grammar
  - If the input does not match the grammar, it fails
- **Formally specify the protocol grammar**
  - Define what is legal, and what is not, and write this down in a machine checkable manner
    - Define the grammar in as restrictive a manner as possible – e.g., don't use a Turing-complete parse when a regular expression will suffice
  - Specify what happens if the input data does not match the grammar: what causes a failure? How is it handled?

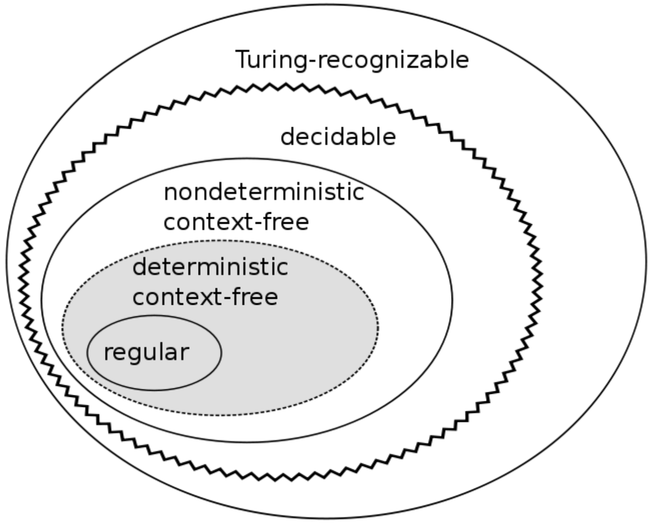


Source: http://langsec.org/papers/Sassaman.pdf

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Generate the Parser (1/3)

- Difficult to reason about ad-hoc, manually written, parsing code
  - It performs low-level bit manipulation, string parsing, etc., all of which are hard to get correct
  - It tends to be poorly structured, hard to follow
- Rather, **auto-generate the parsing code**:
  - If input language is regular, use regular expression
  - If input language is context free, use a context free grammar
  - If neither of these work, use a more sophisticated parser, with minimal computational power
  - Generate strongly typed data structures, with explicit types to identify different data items
- Focus on parser correctness and readability
  - **Parsing performance matters less than security**



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Generate the Parser (2/3)

- Use existing, well-tested, parser libraries
  - For Rust code, use **`nom`** or **`combine`**:

    nom, eating data byte by byte

    https://github.com/Geal/nom

    Combine: A parser combinator library for Rust

    https://github.com/Marwes/combine

  - For C or C++, use **`hammer`**

    Hammer: Parser combinators for binary formats, in C. Yes, in C. What? Don't look at me like that.

    https://github.com/UpstandingHackers/hammer



2017 IEEE Symposium on Security and Privacy Workshops

Writing parsers like it is 2017

Pierre Chifflier
Agence Nationale de la Sécurité
des Systèmes d'Information

Geoffroy Couprie
Clever Cloud

***Abstract*—Despite being known since a long time, memory violations are still a very important cause of security problems in low-level programming languages containing data parsers. We address this problem by proposing a pragmatic solution to fix not only bugs, but classes of bugs. First, using a fast and safe language such as Rust, and then using a parser combinator. We discuss the advantages and difficulties of this solution, and we present two cases of how to implement safe parsers and insert them in large C projects. The implementation is provided as a set of parsers and projects in the Rust language.**

I. Introduction

In 2016, like every year for a long time, memory corruption bugs have been one of the first causes of vulnerabilities of compiled programs [1]. When looking at the C programming language, many errors lead to memory corruption: buffer overflow, use after free, double free, etc. Some of these issues can be complicated to diagnose, and the consequence is that a huge quantity of bugs is hidden in almost all C software.

Any software manipulating untrusted data is particularly exposed: it needs to parse and interpret data that can be controlled by the attacker. Unfortunately, data parsing is often done in a very unsafe way, especially for network protocols and file formats. For example, many bugs were discovered in media parsing libraries in Android [2], leading to the possible remote exploitation of all devices by a simple MMS message. Today, many applications embed a lot of parsers that could be targeted: web browsers like Firefox or Chrome, media players (VLC), document viewers, etc.

Ironically, security tools such as intrusion detection systems or network analyzers suffer from the same problems, making the security tools a possible and interesting target for an attacker. For example, Wireshark had 95 vulnerabilities in 2016 that could crash the application or be exploited.

As a result, most programs written in C are unsafe, especially in the parts of the code parsing data. For example, more than 500 vulnerabilities on XML parsers are listed in the US National Vulnerability Database (NVD). Even for simple formats like JSON, it's hard [3]: some parsers crash, others have bugs, and many of them give different results because of a different interpretation of the specifications.

The Cloudbleed [4] vulnerability, caused by a bug in a HTML parser written in C, caused sensitive data to leak, and is estimated to impact more than 5 million websites. This bug was possible because of the use of an unsafe programming language, despite using a parser generator (Ragel).

In this paper, we propose a pragmatic solution to write robust parsers and significantly improve software security. First, we show how changing the programming language can solve most of the memory-related problems. Second, we show how parser combinators both help prevent bugs and create faster parsers. We then explain how this solution was implemented in two different large C programs, VLC media player and Suricata, to integrate safe parsers by changing only a small amount of code.

II. Current Solutions, and how to go further

A. Partial and bad solutions

Many tools, like fuzzing or code audits, come too late in the development process: bugs are already present in the code. Instead, developers should focus on solutions allowing them to prevent bugs during development.

Some are trying to improve quality by integrating automated tests during the development process. The devops trend encourages pushing code into production as fast as possible, minimizing the delay by relying on automated tests to assess security. It is important to be agile and be able to fix bugs very quickly. However, the problem with that approach is that it will catch only known (tested) bugs, and will never be exhaustive: this is more compliance testing than security tests. While it can protect against most regressions, it tends to create a false sense of security.

B. Reduce Damage

Hardening methods can be applied to restrict the software to its intended behavior, or limit the consequences in case of successful attacks. While these mechanisms significantly improve the security of the software and the system, they will not fix the initial problem: the restricted application manipulating data is still vulnerable and affected by all the parsing problems. Still, they are interesting, and applications should consider using them as much as possible.

Compiler and OS offer hardening functions that can be used. Some of them can be applied without modifying the software, but the others require a modification or recompilation. We list here only categories of system hardening features.

a) Compiler functions: The compiler can apply hardening functions during the compilation: randomization of the address space (ASLR), stack protector, marking sections and the relocation table as read-only, etc. These functions are activated by compilation flags, and will also trigger build errors on some vulnerable code patterns.

© 2017, Pierre Chifflier. Under license to IEEE.
DOI 10.1109/SPW.2017.39

P. Chifflier and G. Couprie. Writing parsers like it is 2017. IEEE Workshop on Language-Theoretic Security, San Jose, CA, USA, May 2017. https://dx.doi.org/10.1109/SPW.2017.39

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Generate the Parser (3/3)

```
#[derive(Debug,PartialEq,Eq)]
pub struct Header {
  pub version: u8,
  pub audio:   bool,
  pub video:   bool,
  pub offset:  u32,
}


do_parse!(
            tag!("FLV") >>
  version: be_u8        >>
  flags:   be_u8        >>
  offset:  be_u32       >>
  (Header {
      version: version,
      audio:   flags & 4 == 4,
      video:   flags & 1 == 1,
      offset:  offset
  })
)
);
```

Source: P. Chifflier and G. Couprie. Writing parsers like it is 2017. IEEE Workshop on Language-Theoretic Security, San Jose, CA, USA, May 2017. https://dx.doi.org/10.1109/SPW.2017.39

- Specify the types into which parsed data is stored
- Describe the parser using an appropriate formal language – example uses **nom**
- Generate the parser from that language

- Parsing is performed first, and either succeeds in its entirety or fails – the rest of the code uses only safe, pre-parsed, data

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Define Parsable Protocols

- If designing network protocols, consider ease of parsing
  - Minimise the amount of state and look-ahead required to parse the data
  - Prefer a predictable, regular, grammar to one that saves bits at the expense of complex parsing
    - Networks get faster, security vulnerabilities remain
    - The benefit of saving a few bits gets less over time
    - The benefit of being easy to parse securely remains

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Parsing and LangSec

- **Read:** “The Bugs We Have to Kill”
  - Is this approach realistic?
  - Can we combine better parsing with modern, strongly typed, languages to improve network security?
  - Is performance good enough?
  - Can we improve the way we design protocols?



SECURITY

The Bugs We Have to Kill

SERGEY BRATUS, MEREDITH L. PATTERSON, AND ANNA SHUBINA

Sergey Bratus is a Research Associate Professor of computer science at Dartmouth College. He sees state-of-the-art hacking as a distinct research and engineering discipline that, although not yet recognized as such, harbors deep insights into the nature of computing. He has a PhD in mathematics from Northeastern University and worked at BBN Technologies on natural language processing research before coming to Dartmouth.
sergey@cs.dartmouth.edu

Meredith L. Patterson is the founder of Upstanding Hackers. She developed the first language-theoretic defense against SQL injection in 2005 as a PhD student at the University of Iowa and has continued expanding the technique ever since. She lives in Brussels, Belgium.
mlp@upstandinghackers.com

Anna Shubina is a Research Associate at the Dartmouth Institute for Security, Technology, and Society and maintains the CRAWDAD.org repository of traces and data for all kinds of wireless and sensor network research. She was the operator of Dartmouth's Tor node when the Tor network had about 30 nodes total.
ashubina@cs.dartmouth.edu

The code that parses inputs is the first and often the only protection for the rest of a program from malicious inputs. No programmer can afford to verify every implied condition on every line of code—even if this were possible to implement without slowing execution to a crawl. The parser is the part that is supposed to create a world for the rest of the program where all these implied conditions are true and need not be explicitly checked at every turn. Sadly, this is exactly where most parsers fail, and the rest of the program fails with them. In this article, we explain why parsers continue to be such a problem, as well as point to potential solutions that can kill large classes of bugs.

To do so, we are going to look at the problem from the computer science theory angle. Parsers, being input-consuming machines, are quite close to the theory's classic computing models, each one an input-consuming machine: finite automata, pushdown automata, and Turing machines. The latter is our principal model of general-purpose programming, the computing model with the ultimate power and flexibility. Yet this high-end power and flexibility come with a high price, which Alan Turing demonstrated (and to whose proof we owe our very model of general-purpose programming): our inability to predict, by any general static analysis algorithm, how programs for it will execute.

Yet most of our parsers are just a layer on top of this totally flexible computing model. It is not surprising, then, that without carefully limiting our parsers' design and code to much simpler models, we are left unable to prove these input-consuming machines secure. This is a powerful argument for making parsers and their input formats and protocols simpler, so that securing them does not require having to solve undecidable problems!

Parsers, Parsers Everywhere

To quote Koprowski and Binsztok [1]:

> Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation. For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

So do the lower layers of the network stack down to the IP and the link layer protocols, and also other OS parts such as the USB drivers [2] (and even the hardware: turning PHY layer symbol streams into frames is parsing, too!). For all of these core protocols, we add, their parsers have had a long history of failures, resulting in an Internet where any site, program, or system that receives untrusted input can be presumed compromised.

S. Bratus, M. L. Patterson, and A. Shubina. The bugs we have to kill. ;login:, 40(4):4–10, August 2015. http://langsec.org/papers/the-bugs-we-have-to-kill.pdf

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Parsing and Network Security

- Postel’s law
- Parsing

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow