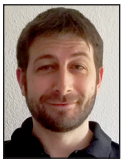
Safe Parsers in Rust

Changing the World Step by Step

GEOFFROY COUPRIE AND PIERRE CHIFFLIER

Geoffroy Couprie handles security and quality assurance at Clever Cloud, develops in Rust, and researches on parser security at VideoLAN. He thinks a lot about cryptography, protocol design, and data management.
contact@geoffroycouprie.com



Pierre Chifflier is the head of the intrusion detection lab (LED) at ANSSI (French National Information Security Agency). He is interested in various security topics such as operating systems, boot sequence, compilers and languages, and new intrusion detection methods, and he is also trying to link all these topics by improving detection tools, writing safe parsers, and deploying tools in a secure architecture. Pierre is also a Debian Developer and has been involved in free software for a long time.
chifflier@wzdftpd.net

Parsers are critical parts of applications, exposed to potentially malicious data but also plagued by the same bugs over a period of years, like memory-related problems. Solutions exist but are often not adopted: many of them require rewriting entire software packages. We describe how to leverage Rust's safety features and close integration with C, the strength of the nom [1] parser combinators library, along with a thorough methodology [2] to make existing software much more secure by rewriting critical parts. By surgically replacing functions, we intend to initiate a change towards robust and memory-safe parsers.

A large part of our infrastructure is built on a sand castle. We have been reusing the same code for decades, the same libraries written in the '90s, the same applications, the same operating systems. We tried, and are still trying, to maintain them, patching bit by bit, mostly in reaction to published vulnerabilities, sometimes as a proactive effort. But all that old code is slowing us down.

And if that was not enough, to connect those pieces of code to each other, we have pages and pages of unclear, ambiguous specifications for file formats and network protocols. How can you be sure your implementation is correct when some remove features, some add features, others implement them incorrectly, and there are parts that are completely open to interpretation. Let's also mention that incorrect files generated by one broken application often end up supported by everyone else.

Additionally, most of that software has been written in C (sometimes still written in K&R) and involves unsafe practices and insufficient testing.

One could say it is a miracle that all of this has worked this long, but there is no luck in that. It is the result of incremental work of thousands of developers patiently fixing bugs, and system administrators monitoring failing services. But we are losing the race now.

Attackers only get better: what was previously difficult gets simpler, and the tools only get smarter. More vulnerabilities are published every day, while we keep the same old code and the same development practices.

We Cannot Rewrite Everything

Whatever the quality of all that code, we cannot replace it. Software gets reused over and over, with each generation of developers building upon what the previous one built. There's much more churn in hardware than software: hardware gets replaced, software stays. We can write new software with better solutions, but it would not fix the millions of devices currently in place, or the billions of applications actually running. Our only option is to strengthen the sand castle bit by bit until it can weather the storm.

How can we achieve that? Even rewriting application by application or library by library is a Sisyphean task. Most of those projects are written in C, containing 10k to 10m lines of code. Large parts of that are unmaintained, but there's also a huge domain knowledge embedded in the code. Thousands of bug fixes, improvements, and experimentations with the specifications were done over the years. And the developers themselves carry most of this knowledge. Rewriting a project completely means losing that knowledge and hitting most of those bugs the old project solved. In addition, rewriting the project entirely creates political issues and requires teaching the new ways to developers, all while maintaining the old version. This is impossible to do in most cases.

Here is what we propose: there are specific parts of applications and libraries, weaker than the rest, that could be rewritten, while keeping all of the domain knowledge present in the rest of the code. Since file formats and protocols are the point of entry in most applications, we concentrate on the parsers and state machines, an often overlooked and vulnerable part of the code.

The LangSec approach is in changing the way we view software: we usually see our programs as some kind of engine or industrial machine that we set up and monitor but that, except for the occasional button push, largely runs by itself. That vision is flawed: our computers, operating systems, and programs are designed to modify their behavior in complex ways depending on their input.

The data you feed to your code—be it network packets, files, sensor data—drives your code, not the other way around. That specific bit at that specific address in the file determines whether your code goes into the `if` or the `else` of that specific branch. Your application is in fact a virtual machine, and its language is the input data. What can we do with this language? By modeling that input language correctly, or restricting it to a manageable subset, we can greatly reduce the attack surface of our applications in their most vulnerable elements.

If we replace the parsers and protocols in an existing application, we can better protect it from the attackers' point of entry while keeping the most useful parts of the code running. To that end, we need languages and tools that can easily integrate themselves inside a C application.

Choosing the Tools

We decided to use Rust for various reasons: the language is designed to avoid memory vulnerabilities and development issues frequent in other languages. Rust does not use garbage collection; the compiler is smart enough to know when to allocate and deallocate memory. The compiler will complain if the code is unsafe. With this, the compiler can protect your code from common flaws like double free, use after free, adding bounds check to buffers, etc. Rust is even able to know which part of the code owns which part of the memory, and it warns you when your code manipulates data from multiple threads.

Rust has been available for years now (first stable release in May 2015) and has been steadily improving. Because of the focus on the compiler, instead of fixing a memory safety issue in your code, you can improve the compiler so that nobody will ever get that issue again. Do not fix bugs, fix bug classes.

As you learn more Rust, you tend to rely more and more on the compiler to verify the code, instead of keeping track of dozens of pointers in your head, thus freeing you to think about the most valuable parts of the application.

Along with those features, Rust can work at the same level as C applications. There's no runtime. There is no garbage collector (important in time-critical software). It can even work without an allocator. As an example, it can be used for embedded development, from microcontrollers to larger CPUs. To that end, Rust code can easily import C functions and structures and use them natively, but the opposite works as well: you can expose functions and structures to be used by C (or other language) applications. This is a crucial aspect of rewriting C code: sometimes, we have to expose and manipulate the exact same types the target application is using.

Writing parsers manually in Rust is not enough. We can still find bugs, although they are often less critical than the ones you would find in C applications [7]. Parsing software correctly is hard, and anybody can make mistakes.

So we use nom [1], a parser combinators library written in Rust. Parser combinators are an interesting way to handle data. You assemble small functions, like one that recognizes "hello," or one that recognizes alphanumerical characters, and you combine them to make more complex parsers through the use of combinators. There are combinators for lots of cases, like "terminated," that would apply two parsers in a row, then return the result of the first if both are successful, or branching combinators that apply different parsers depending on the result of a first one.

Those parsers are always functions with the same signature, which means even complex parsers can be easily reused in other parsers. You end up writing a lot of small parsers, then you can test them separately, and reapply them in larger parsers as you see fit. An approach based on parsers generated from a grammar, on the other hand, tends to lack flexibility and is harder to test. Such parsers are also quite restrictive in what you can allow from the format you are trying to pass. But since nom parsers are just functions, you can perform whatever complex, ambiguous, dangerous tasks you need to, and as long as the interface is the same, you can plug that parser with other parsers. This is an important property, since most formats are badly designed and can require unsafe manipulations.

The nom library leverages Rust features for performance and safety: since the compiler always knows which part of the code owns which part of the memory, and tracks references properly, nom can work on slices of the original data instead of copying bytes around. In most cases, the parser will only allocate on the stack and be zero-copy [3].

nom has been available for some time now and has been used extensively for various formats and protocols in production software.

Armed with a safe, low-level language, and a parser library, we can now start rewriting core parts of our infrastructure.

How to Replace Part of a C Application

Not all existing applications will easily support a rewrite of their parser. If that part of the code is highly coupled with the rest, it will be problematic. Thankfully, as said earlier, we do not need to rewrite everything. Find a restricted subset of the parser, isolate it, rewrite it, then expand to other parts of the application.

The key is in defining the interface correctly. Deterministic functions are the easiest to replace, and structures are usually the hardest, since multiple parts of the code might use directly internal members of that structure (accessors are not a common practice in C). But there are a lot of tricks one can use to help in the task. As an example, commenting out a member of a structure and launching a build can expose all of the uses of that member, which makes it easier to measure how much work is needed.

When performing a rewrite, you will often need to import C code and expose your Rust code to C. You can write the Rust definitions and the C headers by hand, but Rust has tools to automate this. Rust-bindgen can import C structures and functions from C, and generate Rust bindings. While the generated code might be a bit complex at times, it is a great way to start a project and generate code that you can edit later. The opposite way works as well: you can employ rusty-cheddar to generate C headers.

The missing part for the integration is the linking phase: think of how you will link the Rust part to the C part. Do you make a static or dynamic library? Do you generate an object file that you feed to autotools? The Rust compiler can generate any of those, and they can then be handled by the build system, be it autotools and makefiles, CMake, scons, etc.

On the build-system side of things, Rust uses the cargo package manager to download libraries (called crates), build and link them, and publish new libraries and applications. That tool greatly increases the productivity of Rust developers. Unfortunately, the package management part requires an Internet connection to download packages, which might not be an option (do you expect your makefile to make network calls?). Fortunately, cargo is easy to extend with separate tooling. You can use cargo-vendor or cargo-local-registry to download crates in advance and store them in an archive somewhere. That way, you can freeze the dependency list of an application and make its compilation reproducible, while keeping a simple way to update dependencies when needed.

Start Integrating Some Rust

Once you have the build system set up, you can start actually writing Rust code. We would recommend that you develop the nom parser in a separate crate: that way, you can reuse it in other projects (Rust or other languages), and you can employ Rust's unit testing and fuzzing facilities. Any fuzzing result can then be reused as a test case for your parser.

nom parsers work well on byte slices, a Rust type that contains a pointer and a length. You can easily transform any C buffer to this. They never modify their input, and they don't even need to own it. This is important for integration in C applications: even if we know that Rust code could be stronger than the rest of the application, it is still a guest in someone else's house. If possible, let the host code handle allocations, opening files, etc. This is a really good tip to apply, because libraries with reentrant, deterministic functions without side effects are easy to integrate, and I/O is where most of the errors can happen. This is also a part that (hopefully) has been stabilized long ago in the host application.

The nom parser can return sub slices of the input without copying them and will guarantee that the data is within the bounds. In some cases, it does not even need to see the whole input. As an example, for media formats, you would read a block's header, let nom decide which type of block it is, and the parser would tell you how many bytes of the block you need to send to the decoder.

Here is the code of the TLS 1.3 ServerHello structure definition and message parsing:

```
rust
pub struct TlsServerHelloV13Contents<'a> {
    pub version: u16,
    pub random: &'a[u8],
    pub cipher: u16,

    pub ext: Option<&'a[u8]>,
}

pub fn parse_tls_server_hello_tlsv13draft18(i:&[u8])
    -> IResult<&[u8],TlsMessageHandshake>
{
    do_parse!(i,
        hv:     be_u16 >>
        random: take!(32) >>
        cipher: be_u16 >>
        ext:    opt!(length_bytes!(be_u16)) >>
        (
```

```
            TlsMessageHandshake::ServerHelloV13(
                TlsServerHelloV13Contents::new(hv,random,cipher,
                ext)
            )
        )
    )
}
```

This code generates a parser reading some simple fields, and an optional length-value field for the TLS extensions (not parsed in that example), and returns a structure. All error cases are properly handled, especially incomplete data.

One characteristic of TLS is that the parsing of messages is context-specific: the content of some messages cannot be decoded without having information about the previous messages. For example, the type of the Diffie-Hellman parameters, in the `ServerKeyExchange` message depends on the ciphersuite from the `ServerHello` message. Because of that, the context-specific part is separated from the parsing. A state is used to store the variables, and a state machine is implemented to check that transitions are correct, and also to choose the next parsing function when needed.

The state machine is implemented using pattern matching on the previous state, and the parsed incoming message, to select the new state.

```
rust
match (old_state,msg) {
    // Server certificate
    (ClientHello,         &ServerHello(_))         =>
    Ok(ServerHello),
    (ServerHello,         &Certificate(_))         =>
    Ok(Certificate),
    // Server certificate, no client certificate requested
    (Certificate,         &ServerKeyExchange(_)) =>
    Ok(ServerKeyExchange),
    (Certificate,         &CertificateStatus(_))   =>
    Ok(CertificateSt),
    (CertificateSt,       &ServerKeyExchange(_)) =>
    Ok(ServerKeyExchange),
    (ServerKeyExchange, &ServerDone(_))          =>
    Ok(ServerHelloDone),
    (ServerHelloDone    , &ClientKeyExchange(_))  =>
    Ok(ClientKeyExchange),
    // ...

    // All other transitions are considered invalid
    _ => Err(InvalidTransition),
```

In some cases, the next state depends not only on the message type but also on content. In that case, the packet content is also used in the pattern matching to select the new state.

Finally, the combinator features of nom are especially useful for protocols like TLS: TLS certificates are based on X.509, which uses the DER encoding format. This makes writing an independent parser easier, as in the following code:

```
rust
use x509::parse_x509_certificate;

/// Read several certificates from the input buffer
/// and return them as a list.
pub fn parse_tls_certificate_list(i:&[u8])
    -> IResult<&[u8],Vec<X509Certificate>>
{
    many1!(i,parse_x509_certificate)
}
```

Parsing an X.509 certificate is done by combining the DER parsing functions:

```
rust
pub fn x509_parser(i:&[u8]) -> IResult<&[u8],X509Certificate> {
    map!(i,
        parse_der_defined!(
            0x10,
            parse_tbs_certificate,
            parse_algorithm_identifier,
            parse_der_bitstring
        ),
        |(_hdr,o)| X509Certificate::new(o)
    )
}
```

Be wary of the high coupling that can appear between the parser and the rest of the code in some C applications. This is where most of the work can happen and is usually the result of years of hacks upon hacks to add a feature "quick and easy."

We usually recommend that the parser has a clear interface with the rest of the code, in the form of a list of small, deterministic parsers and a reduced state machine above it: not a complete state machine intertwined with the parsing (as in this http parser [8]) since those are hard to debug and extend, nor a state machine informally implemented via calls from other parts of the code.

The state machine is the main interface for the rest of the code: you feed it data to parse, it decides which parser to apply depending on the current state, changes its state depending on the data that was parsed (if successful), then returns with info to drive the input consumption: how many bytes to consume (or how many more bytes are needed) or to stop consuming if there was an error. You can then query this state machine for the information you want and for data to write back to the network (in the case of a network protocol).

If the code is not highly coupled, you could even rewrite function by function, since the Rust code can expose C-compatible functions. Beware, though: take the time to write a correct internal API for Rust code, since at some point, you might stop exporting those functions and call the underlying functionality directly from Rust.

You could spend a large part of the work making the new parser bug compatible with the old one. This is often a bad approach, since both parsers will probably not recognize the exact same set of files. You only need to worry about recognizing the same representative set of samples. Most C parsers are not even really tested regularly anyway. If you still want to get close results to the original parser, you could employ a smart fuzzer to do the work of testing the difference. Write a program that wraps both the C parser and the new nom one, and that panics if both parsers do not return the same result.

Once the parser is written and in the source, be happy, for now the "interesting" part of the work will begin: getting it accepted in the tree and deciding how you will handle the software suddenly requiring a Rust compiler along with the old C toolchain.

Going Further

This approach of surgically rewriting parts of an application works well since it is designed to have a minimal impact on the original project. It can be used as a stepping stone to start replacing larger parts of the application once all the details of build systems and developer training are handled.

But some projects could never handle that kind of precise touch. Some libraries, still in active use today, have highly coupled spaghetti code, relying heavily on GOTO or setjmp, and are basically untested and unmaintained. This is one of the rare cases where we'd recommend rewriting the whole project in Rust. This is a place where this language can shine; you could write a whole new library, completely API-compatible with the old one, that you could drop into package managers as an alternative.

Think of how many parts of our infrastructure we could replace like this, bit by bit. It's a Herculean task, so we need to start now.

This work was presented in the 2017 LangSec Workshop [4], in the "Writing parsers like it is 2017" [2] paper. The parsers and tools are published in the Rusticata [5] and VLC module [6] GitHub projects.

References

[1] Rust parser combinator framework: https://github.com/Geal/nom.

[2] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," IEEE LangSec Workshop '17: http://spw17.langsec.org/papers/chifflier-parsing-in-2017.pdf.

[3] G. Couprie, "Nom, a Byte-Oriented, Streaming, Zero-Copy Parser Combinators Library in Rust," IEEE LangSec Workshop '15: http://spw15.langsec.org/papers/couprie-nom.pdf.

[4] IEEE LangSec Workshop'17: http://spw17.langsec.org.

[5] Rusticata: Safe parsers community: https://github.com/rusticata.

[6] Helper library to write VLC modules in Rust: https://github.com/Geal/vlc_module.rs.

[7] List of Rust applications with bugs found by fuzzing: https://github.com/rust-fuzz/trophy-case.

[8] Node.js http parser: https://github.com/nodejs/http-parser.