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.

DOI 10.1109/SPW.2017.39

b) Kernel protections: The kernel of the operating system can provide useful limitations: restricting what a process can see, removing the ability to debug a process, killing the process if anything suspicious is detected, etc. These functions are not all provided by default by the operating system, so adding a patch like grsecurity is necessary to get more features.

c) Behavior restrictions: To reduce the attack surface, a process should be restricted to a minimum set of privileges by dropping its privileges and capabilities. Some mechanisms like Pledge, seccomp or SELinux provide a way to go further and limit a process to only the system calls it should do.

d) Isolation: Applications can be isolated in sandboxes or containers, to limit the interactions they can have with rest of the system. This complicates the ability to compromise other processes, and can also hide the hardware to processes that should not have access to it.

e) : Combined together, these features can be used to build a robust architecture to host applications. A detailed architecture of a hardened network IDS appliance based on these principles has been published in C&ESAR [5].

In addition to these mechanisms, we now propose a solution to fix the software itself.

C. Moving security from the developer to the compiler

After years of code auditing, publishing and teaching good practice and still finding the same problems, we must admit that current methods do not work. To the authors, it mostly points to the necessity of changing one key part of the design of software: the programming language.

Using a safe programming language, the security moves from the developer to the compiler. This ensures that no security check will be forgotten, and that the compiler will be able to use its knowledge of the code (like read-only variables, non-aliased values, memory management etc.) to produce code that is both faster and safer. It also allows to not only fix bugs, but rather to *fix bug classes*.

Obviously, compiler bugs can still happen. However, when fixed, all users and programs will benefit of the fix.

Usually, changing the language means the entire software must be rewritten. This causes several problems: it is not accepted well by the initial software community (changing the language is both an important choice, and affects the group of possible contributors), it would take a long time on large projects, and the visible result would be to get the same features. It also is often perceived as a very negative judgment on the existing code quality, which is not a good way to start contributing to a project.

For these reasons, it seems more reasonable and pragmatic to try to change only a small part of the initial software, without affecting the rest. For example, replacing a parser (ideally, only one function in the initial code) by its safe version, recoded in the new language. By focusing on the critical parts, one can both get faster results, and propose a better contribution.

A great advantage of this method is that using this method, the safe parser is not only useful for one project, but it can be used in other software as well, using the same embedding method.

D. Choice of language

In this section, we describe the reasons for changing the programming language. We assume that the original software is written in C or C++, which is often the case, but the same arguments are also valid to other languages.

A key point here is that the objective is not to rewrite entire projects, but to focus on critical functions or components. This is especially important to ease acceptance by the community of the original software, but comes with a price: the project becomes the sum of components written in different languages.

A study of the intrinsic security characteristics of programming languages [6], [7] has shown that the programming language is not only a tool, it can contribute to (or destroy) the security of the resulting software.

Several languages were tested to implement parsers: OCaml, Go, C Python, Haskell, etc. Only the strongly typed languages were retained, because type safety is an essential quality for security. Indeed, it spots many programming errors during compilation, prevents whole classes of bugs like uncontrolled signed/unsigned conversions. Strong typing forces the developer to explicitly declare all conversions, which is especially important to avoid unwanted behaviors. It also allows to define an API for functions that cannot be bypassed: by enforcing strict rules, it restricts the developer to use functions and arguments only in the expected way.

Languages with garbage collection have also been removed, because they come with many problems: some of them need to stop the entire program when collecting (killing performance), most of them do not work properly with multi-threading programs. Even when optimized, this leads to difficult balance between high latency and reduced throughput plus unpredictable heap growth [8]. Memory management can be a big deal, especially for situations where the programs need to be memory bound. For example, Dropbox rewrote the memory critical components of the Magic Pocket application from Go to Rust [9] to be able to control the memory footprint of the application.

Often, the C language is chosen because it is fast, and has a memory model close to the low-level hardware. It is very important, when introducing components in a new language, not to kill the performance. This has two consequences: the produced code must be efficient, but this also implies that the memory model of the two languages must be close (ideally, it would be directly compatible, or zero-copy). Indeed, if the memory models are different, every data exchange will require a format change or a copy, which can become very slow.

None of the previous languages could fulfill all the requirements. The differences and possibilities of all possible languages will not be debated here, as this would lead to a potentially infinite discussion.

We chose the Rust language, because of the following properties:

- Managed memory: Rust has a concept of *lifetimes* and *ownership*, which guarantees the safety of memory without requiring a garbage collector or reference counting, at compilation time. In particular, Rust prevents use after free and double free flaws
- No garbage collector: controlled memory use and life of objects
- Thread-safe: this is guaranteed by the compiler
- Strong typing: as described above, this property is very important
- Efficient: the source is compiled to native code, and produced code that tends to be quite fast (similar to C)
- Zero-copy: data buffers (called slices), objects and references can point to the same locations, and the compiler uses it to avoid copying data while ensuring the memory safety
- Easy integration with C, as few data conversions as possible
- Clear marking of the safe/unsafe parts of the code: all the potentially dangerous instructions for memory safety (including calls to C functions and syscalls) must be enclosed in an unsafe block, and the rest is statically verified to be safe
- Good/large community: it's important to use a good language, but it is better to have active and helpful users.

The properties listed here are not the exhaustive list of Rust language properties, but only the most interesting to solve our problem. Another language than Rust could have been used if providing equivalent properties.

In addition to the language properties, it is interesting to consider the security properties of the produced binary: does the compiler add checks ? Does it use the security features of the operating system, like randomization ? The generated code (in LLVM form) has been analyzed to verify some of these points (especially the memory safety and runtime security features). The results are presented in the appendix.

Changing the language is important, but is not enough. Another component is required to help fix logical bugs, and make sure parsers are both easier to write and do not contain errors.

III. Parser combinators

A. Handwritten parsers VS parser libraries

Most of the low-level parsers found in the wild are written entirely by hand. There are a few reasons for this: a lot of parser libraries focus on textual languages and ignore binary formats, and one can usually get better performance by describing the state machine manually.

Unfortunately, writing a correct parser while managing data streaming in the same code is error-prone, and often leads to unmaintainable code, or at least needlessly complicated code. The classical solution to write manually a state machine [10] that advances depending on the current byte is usually *written once*: any modification to the transitions can affect many different parts of the parser, thereby introducing bugs.

There is a kind of parser library often employed when the developer has taken a *Compilers 101* course: parser generators based on a textual grammar describing the format, like *lex* or *Ragel* [11]. Those tend to produce parsers of good quality, since most of the parsing bugs are already handled by the code generator. The drawbacks lie in writing the grammar the right way to avoid ambiguities and trying to fit context sensitive formats in libraries that may not support them.

B. Parser combinators

Parser combinators propose a middle ground between handwritten parsers and generated parsers. They are made of small functions that handle basic parts of a format, like recognizing a word in ASCII characters or a null byte. Those functions are then composed in more useful building blocks like the pair combinator that applies two parsers in sequence, or the choice combinator that tries different parsers until one of them succeeds. Those can be combined to make format specific parsers to recognize meaningful elements like a file header or a text line.

They provide an interesting alternative because the combinators already handle safe data consumption between different parsers, and since they are functions like one would write in any programming language, it is easy to write our own low-level building blocks and integrate them in larger parsers.

Parser combinators come from the functional programming world, and as such, they get a crucial property: the functions must be completely deterministic, and hold no mutable state. A parser will take data and parameters as arguments, and return the parsed value and remaining data if successful, or an error. Calling a parser with the same input always gives the same result, and it will never modify the input data.

They are traditionally better suited for garbage collected languages, since passing data from one parser to the next can involve a lot of shallow copies. Still, we will show in the next section that there are solutions for lower level languages.

The approach based on small functions composed in larger parsers also makes them easily testable in isolation, and simpler to maintain. It is then possible to write tests or fuzzers for specific parts of a parser without creating a parser state or loading a complete file in memory.

They are still not enough to write a complete format parser, as they will not manage data accumulation. But their deterministic behavior simplifies writing the state machine: it calls the parser on the currently available data, makes a transition if the parser returned a value or an error, or stay at the same state and accumulates more data in the parser indicates that more data is needed.

Thanks to this approach, they are suited both for parsing contiguous data that fits completely in memory, and streaming protocols.

C. nom

nom [12] is a parser combinators library written in Rust. It takes inspiration from Haskell's Parsec and OCaml's menhir libraries. They are written mainly with Rust macros to generate safe data consumption code.

The underlying idea of nom stems from Rust's memory handling. Since the compiler always knows which part of the code owns which part of the memory, and Rust has a *slice* type (a wrapper over a pointer and a size) that represents data borrowed from somewhere else, it is possible to write a parser that will never copy input data. It will instead only return generated values or slices of its input data. This approach has the potential to make efficient parsers.

Additionally, the parsers do not require ownership of the data, which makes them suitable to work inside C applications. They can work on non-contiguous data types like ropes [13].

Most of the parsers are written using macros abstracting data consumption, but it is possible to write specific parts of a parser manually, as long as the function follows this same interface:

```
fn parser<Input, Output, Error>(input: Input) -> nom::
    IResult<Input, Output, Error>;

// with IResult defined like this:
#[derive(Debug,PartialEq,Eq,Clone)]
pub enum IResult<I,O,E=u32> {
   /// indicates a correct parsing, the first field
       containing the rest of the unparsed data, the second
        field contains the parsed data
  Done(I,O),
  /// contains a Err, an enum that can indicate an error
      code, a position in the input, and a pointer to
      another error, making a list of errors in the parsing
       tree
  Error(Err<I,E>),
  /// Incomplete contains a Needed, an enum than can
      represent a known quantity of input data, or unknown
  Incomplete(Needed)
}
```

Listing 1. nom parsers interface

While it is not recommended, for security reasons, to write whole sections manually, this function signature allows writing context sensitive parts, or code that would not fit well in a more theoretical framework. Like the *unsafe* keyword in Rust, we can isolate a dangerous part of the code and integrate it in safer code, as long as it follows a contract with the compiler for Rust, or with the parser interface for nom.

The common combinators are supported: *take!* selects a configurable amount of bytes or characters, *pair!* applies two parsers in a row and returns a tuple, *many1!* applies a parser one or more times and returns a vector of values, *preceded!* applies two parsers and returns the result of the second one, etc. Along with those combinators, nom provides facilities to integrate regular expressions for regular subsets of the format, automatically handles whitespace separated formats, and supports bit level parsing for low-level formats.

Thanks to Rust's close relationship with C, it is possible for nom to parse data directly to C compatible structures, allowing for easy integration in C and other languages.

Finally, it accepts as input byte slices, UTF-8 encoded strings, can handle and can be extended to support more input types, as long as they implement a set of traits that the parsers use.

On the performance side, the design of the parsers makes for very linear code, a long list of branches, and the parser state is represented through stack frames between parser calls. This result differs from the traditional switch-based state machine approach, where the current byte indicates the next state and where to jump in the code. While a carefully written goto-based state machine can get nearly optimal performance, the code generated by nom has various benefits out of the box. The parsers work well on contiguous data, and the corresponding code tends to be contiguous. At the CPU level, that code is more cache friendly than a switch-based state machine jumping everywhere in the code, and can make branch prediction easier. In measurements, nom can get out of the box a performance in the same range as carefully written C parsers [14]. Further optimizations can then be applied as needed, by writing custom functions.

Since the parsers are stateless and do not need ownership of the input data, it is relatively easy to write the parser in nom, add unit tests in Rust, then call into Rust from C. The C side only needs to understand the three return states: yield a value (and consume a part of the input), an error was encountered, or we need more data. The wrapping code is then responsible for any data read or buffer reallocation, and those will not be affected by the parser.

As we will see in the next sections, integrating a Rust parser in a C program is straightforward.

IV. Integrating Rust parsers in an existing application with minimal impact

As we try to replace parts of larger C or C++ applications with Rust elements, it is crucial to make the integration as painless as possible, to prevent objections from existing developers. If the integration required rewriting a large part of the orchestration code, we would probably end up adding more bugs than the ones we solve.

Thankfully, the parsers are a good target for a rewrite. They are rarely modified, as formats and protocols do not change too often. Infrequent modifications of unfamiliar parts create messy code that nobody wants to maintain. They are often well contained in the IO handling code, do not modify directly the application's state, and hold a very small state themselves.

We shall note that replacing C parsers by a safer alternative will not fix bugs in the rest of the code. Rust can only guarantee memory safety up to the interface with the rest of the application. While it is possible to rewrite the whole host application, it is beyond the scope of this paper. We are trying to rewrite the parsers, which are often the weakest part of a program, without impacting too much the hosting code. We will see that replacing C code with Rust is a matter of keeping consistent interfaces and complying with build systems.

A. Interfaces

If it is possible to keep the exact same interface as the C code, a successful approach consists in wrapping the C code in a Rust project (it is possible with the *cargo* build tool) and rewrite the functions one by one. We saw that was not the right way for parsers, since they would be rewritten in a completely different style. Still, a Rust project can expose C compatible functions and even emulate the C style to communicate with the host program.

We recommend that the parser be written and tested in a separate Rust project, to facilitate testing and reuse in other applications. The interface code is then tasked with transforming unsafe C types like pointers to void, to richer Rust types, and the reverse when returning values to C. This approach also works when integrating with other languages like Java with JNI or JNA.

The Rust code must appear as a set of deterministic, reentrant functions from the C side to facilitate usage. In some cases, we want to keep some state around. If needed, we can wrap a Rust value in a *Box* that will be passed to the C code as an opaque pointer, for which we provide accessor functions. Since Rust can apply the *repr(C)* attribute to data structures to make them accessible from C code, we could avoid the opaque pointer, but it gives stronger guarantees over referencing internal data that could be deallocated at any moment by the Rust code.

On the allocations side, the Rust code can work with its own allocator instead of the host application. In cases where it is not wanted, one can build a custom allocator calling the host one. In the end, this is not an important issue here, since nom mainly uses the stack and returns slices to data allocated by the host code.

To further reduce its impact, we can make sure that the Rust part never owns the data to parse, by only working on slices, and returning slices. That way, the only important values moving back and forth are pointers and lengths. This is especially interesting for media parsers where we parse a small header, then return a slice to a part of the data we never need to see.

B. *Build systems*

Integrating the Rust code in the build system is a large part of the work needed. While it is not especially hard, it requires fixing a lot of details, like passing the platform triplets between compilers, setting up the right paths for intermediary or final objects, ordering the build of components and linking them.

We found three ways to integrate Rust projects. The first consists in making a static library and C headers that will be used like any other library in the C code. This is by far the easiest way, if the application can work that way. For our examples in VLC media player and Suricata, making a dynamic library that emulates the behavior of C dynamic libraries makes a well-contained process, as long as we can assemble everything in the right folders. In the case of VLC media player, we ended up building the module as an object file to let the build system take care of linking libraries.

V. Example: a FLV demuxer for VLC media player

The goal of VLC media player [15] is to be able to play every existing video format or protocol. As such, it embeds parsers for a lot of different formats, and those parsers are mainly written in C or C++. Multiple security vulnerabilities were found in those parsers over the past few years [16].

The video container formats and the streaming protocols tend to be complex and ambiguous. They evolved organically from the needs of different companies and will usually trade convenience for decoding performance.

VLC media player is then a good target for experimentation of Rust parsers. The goal of the project is to integrate a FLV parser as a VLC plugin.

A. *Writing a FLV demuxer*

The format [17] we chose is quite simple. It contains first a file header, then a list of audio, video or metadata packets, each being part of an audio or video stream.

It can contain 12 audio codecs and 9 video codecs, with specific parameters, and time synchronization information.

A demuxer usually has two roles. First, it must decide if it can parse the stream. Media players will usually test a few demuxers on a file, then use the one that recognized the file. Then, for each new data packet, it must obtain the following information:

- to which stream this packet belongs to
- which codec is used
- when it should be presented to the user
- which parts of the data stream contains the encoded audio or video

1) Writing a FLV parser with nom: The nom FLV file parser, called flavors [18], declares separately each packet header to parse (file header, audio packet, video packet, etc.). This is done that way for two reasons:

- each header parser can be written and tested independently
- the parser will typically only see the header's part of the input, and let the calling code handle the encoded data

The code follows a process where we first declare a structure that will contain the parsed data, then we use the *named!* nom macro to declare a parser returning that structure. In the following example, the file header begins with the *”FLV”* characters, then a byte encoding the version number, then a flag byte indicating if there is audio or video data, and a big endian 32 bit unsigned integer for the offset at which the data may begin.

We used the *do_parse!* combinator to apply multiple parsers in a row and aggregate the results. Partial data consumption is done automatically: the first parser will consume the *”FLV”* characters, the next parser will take the next byte and store it in *version*, and so on and so forth. If there was not enough data, the parser would indicate how much more data it needs to advance to the next step.

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

```
named!(pub header<Header>,
  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
    })
  )
);
```

Listing 2. FLV file header parsing

Rust's unit testing facilities and nom's approach focused on slices helps in writing tests for our parsers. In the following example, we can use the *include_bytes!* feature to include directly into the executable a test file, and refer to it as a byte slice. Of course, the file is only present in the unit tests, and would not be included by the library in another project.

The file header parser can then be tested on a subslice of the first nine bytes of the file. We can refer to other subslices by index for other parsers as well.

```
#[cfg(test)]
mod tests {
  use super::*;
  use nom::IResult;

  const zelda : &'static [u8] = include_bytes!("../assets/
      zelda.flv");

  #[test]
  fn headers() {
    assert_eq!(
      header(&zelda[..9]),
      IResult::Done(
        // the parser consumed the whole slice
        &b""[..],
        Header { version: 1, audio: true, video: true,
             offset: 9 }
      )
    );
  }
}
```

Listing 3. FLV file header unit testing

B. VLC Media Player Architecture

To accommodate the long list of formats, codecs, and input or output devices on multiple platforms, VLC media player is built with a main library called libvlccore, that handles the media pipeline, synchronization, loading modules. The codecs, demuxers and other parts are built as plugins, dynamic libraries that libvlccore loads depending on the media that should be played, recorded or converted, as seen in figure 2.

Those modules just need to export some C functions that will be called by libvlccore to determine the module's type and get useful metadata as C structures and function callbacks.

Since Rust can reproduce most of C's calling conventions, it is relatively easy to create a dynamic library that directly emulates a VLC plugin's behavior.

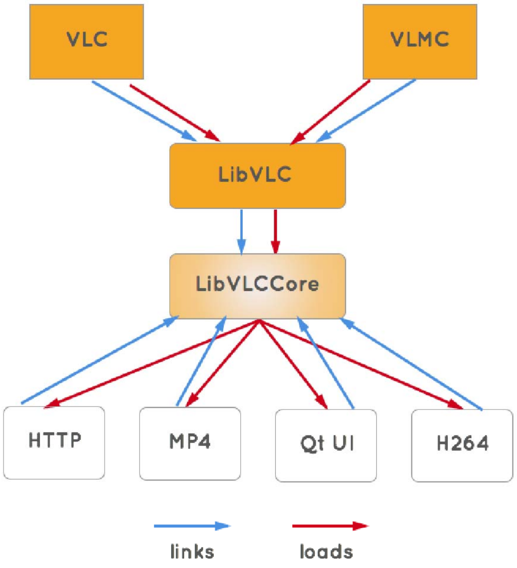

Figure 1. Plugin architecture for VLC media player

C. Integrating Rust code in a C application

1) Writing bindings: The first step in writing Rust code to insert inside a C program is to make bindings to the required functions. In the case of VLC, libvlccore provides a number of helper functions to manipulate streams or metadata.

Rust is well suited to write those bindings manually, as seen in the following listing, but due to the large API provided by libvlccore, we used rust-bindgen [19] to generate those bindings automatically from C headers.

```
// structure definitions
#[repr(C)]
pub struct vlc_object_t {
  pub psz_object_type: *const c_char,
  pub psz_header:      *mut c_char,
  pub i_flags:         c_int,
  pub b_force:         bool,
  pub p_libvlc:        *mut libvlc_int_t,
  pub p_parent:        *mut vlc_object_t,
}

// function imports
#[link(name = "vlccore")]
extern {
  pub fn stream_Peek(stream: *mut stream_t, buf: *mut *
       const uint8_t, size: size_t) -> ssize_t;
  pub fn stream_Read(stream: *mut stream_t, buf: *const
       c_void, size: size_t) -> ssize_t;
  pub fn stream_Tell(stream: *mut stream_t) -> uint64_t;
}
```

Listing 4. Manual declaration of structures and import of functions

2) Exporting C functions from Rust: Once the bindings are generated, the Rust code can call into the C API, and we now need the C code to call into Rust. A VLC module must export a *vlc_entry_<VERSION>* function that libvlccore will call. This function declares the module's name and description, its capabilities, and callbacks to create or close a module's context.

Rust can declare and export functions with the same interface as C code. It was not necessary for this project, but it is possible to generate C headers for Rust code automatically with rusty-cheddar [20].

3) Parsing data: A VLC demuxer will call the *stream_Peek* or *stream_Read* methods to access the beginning of the data and try to parse it. An important pattern appears there: the Rust code never owns the data, it is always passed by the C code. Since nom can work on immutable slices, it parses the data it is given, preferably in a stack allocated array, and has no influence on the file or network I/O.

In the following example, we read one byte of the input inside a stack allocated array and try to parse an audio packet header. If the parser was successful, we initialize codec structures from the parsed values (here, an audio frequency), and requests from libvlccore a new block of data of the size we just parsed. This Rust VLC plugin replaces the exiting FLV demuxer, there is no need to reparse the data afterwards. The data block is managed by libvlccore, and will go directly to the decoder once the demuxer ends this parsing phase.

```
let mut a_header = [0u8; 1];
let sz = stream_Read(p_demux.s, &mut a_header);
if sz < 1 {
  return -1;
}

if let nom::IResult::Done(_, audio_header) = flavors::
     parser::audio_data_header(&a_header) {
  if ! p_sys.audio_initialized {
    es_format_Init(&mut p_sys.audio_es_format,
                   es_format_category_e::AUDIO_ES,
                   audio_codec_id_to_fourcc(audio_header.
                        sound_format));

    p_sys.audio_es_format.audio.i_rate = match audio_header
         .sound_rate {
      flavors::parser::SoundRate::_5_5KHZ => 5500,
      flavors::parser::SoundRate::_11KHZ  => 11000,
      flavors::parser::SoundRate::_22KHZ  => 22050,
      flavors::parser::SoundRate::_44KHZ  => 44000,
    };
    p_sys.audio_initialized = true;
  }

  let p_block: *mut block_t = stream_Block(p_demux.s, (
       header.data_size - 1) as size_t);
  if p_block == 0 as *mut block_t {
  vlc_Log!(p_demux, LogType::Info, PLUGIN_NAME, "could not
       allocate block");
    return -1;
  }
}
```

Listing 5. Parsing an audio packet

Since the demuxer only relies on stack allocated arrays and buffers managed by libvlccore, this minimizes the Rust code's footprint: only the header data moves between C and Rust.

4) Finalizing integration: In a large project like VLC media player, one of the biggest challenges in including code from a new language is the build system, based on autotools. We tried various solutions, and in the end, we asked the Rust compiler to generate object files that will be linked by libtool. While the Rust compiler is able to generate complete dynamic libraries, this way was the easiest to integrate with autotools.

To help further in developing VLC plugins, most of the FFI code was extracted in a Rust library [21] that can, amongst other features, automatically generate entry functions and bindings to libvlccore's API.

VI. Example: a safe TLS parser for Suricata

Suricata [22] is an open source network intrusion detection system (IDS). It inspects the network traffic, decodes and analyzes many protocols to apply detection rules.

A network IDS is obviously exposed to all kinds of traffic, especially malicious one. As it implements many protocol decoders in C, this creates a paradox: the IDS is an interesting target for an attacker, because it is likely to have vulnerabilities. Thus, it is a good candidate for adding safe parsers.

The objective is not to rewrite the project, but only to focus on critical parts (the parsers) to rewrite them securely, and integrate easily with the existing code.

The modifications are done in two steps: first, a parser for the TLS protocol was written in pure Rust, independently of Suricata. Then, another small library is added to integrate this parser in Suricata. In this section, we describe the standalone TLS parser.

A. A standalone TLS parser

TLS is a good example of a very important protocol, for which the existing implementations all have problems [23]. Amongst the 51 vulnerabilities in 2014, 15 are directly related to memory safety violations, and 3 to certificate parsing [24]. To cite only one of the well-known vulnerabilities, Heartbleed [25] is a buffer overflow in an unimportant feature, with critical consequences, that could have been prevented by using a memory-safe language.

Even the recently added implementations like BearSSL had the same kinds of issues [26], which clearly shows that even carefully written projects will still lead to the same memory problems.

Even when it's not parsers, the state machine of TLS is also often badly implemented [27].

a) Implementing the TLS parser: A parser for the TLS protocol has been implemented. It covers all the record and message types defined for TLS version 1.2 and the multiple RFCs extending it (resuming session, elliptic curves, client and server extensions, etc.). It also covers TLS 1.3, in the latest draft available (draft 18).

By using a parser combinator, the parsing code is very close to the description of the corresponding structure in the specification. This reduces the risk of mistakes and makes maintenance easier.

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion version;
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<0..2^16-1>;
} ServerHello;
```

Listing 6. ServerHello structure definition in TLS 1.3 IETF draft

Here is the corresponding code of the TLS 1.3 ServerHello message parsing:

```
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)
            )
        )
    )
}
```

Listing 7. Our TLS 1.3 ServerHello message implementation

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.

Note that fields like `version` or `cipher` could be represented as enum values, which would make it even closer to the specifications. However, this is a deliberate choice: malicious data can be invalid, and so invalid values must be accepted by the parser, but detected as invalid.

We solve this problem by adding accessor functions, that returns an `Option` type, being either the enumerated value, or an error. In case of error, the invalid value is still accessible.

```
#[repr(u16)]
pub enum TlsVersion {
    Ssl30 = 0x0300,
    Tls10 = 0x0301,
    Tls11 = 0x0302,
    Tls12 = 0x0303,
    Tls13 = 0x0304,

    Tls13Draft18 = 0x7f12,
}

impl<'a> TlsServerHelloV13Contents<'a> {
    pub fn get_version(&self) -> Option<TlsVersion> {
        TlsVersion::from_u16(self.version)
    }
}

// [...]

let opt_vers = server_hello.get_version();
```

Listing 8. Example accessor function

One other 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, some variables are extracted and stored in a parser context, associated to every connection, and passed to higher-level parser functions.

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 independent parser easier, for example as in the following code:

```
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)
}
```

Listing 9. Combining TLS and X.509 parsers

b) Fragmentation: When writing a TLS parser, special attention is required to deal properly with fragmentation. Indeed, in addition to classical IP fragmentation and TCP chunks fragmentation, TLS defines an application layer fragmentation, combined with the concatenation of multiple messages.

In short, one or more TCP chunks can contain one or more TLS *records*, each of them containing one or more TLS *messages*. A message can be split across multiple records, and a record can be split in multiple TCP chunks.

With a parser in C, that means a lot of pointers arithmetic, and risks of errors. In the proposed TLS parser, parsing is split in two steps: one to ensure we have enough data for a record, and the second to parse a record knowing it is complete. The code to handle fragmentation is both very simple, fast and safe:

```
// Check if a record is being defragmented
let record_buffer = match self.buffer.len() {
    0 => r.data,
    _ => {
            v = self.buffer.split_off(0);
            v.extend_from_slice(r.data);
            v.as_slice()
    },
};
// [...]
match parse_tls_record_with_header(record_buffer,r.hdr) {
```

Listing 10. Record defragmentation

This code roughly means: if no previous data was buffered, use the new data directly (zero-copy). If not, append new data to previous (copy required, as it would in C). In all cases, no additional memory management is required, and the parser remains efficient, using zero-copy when possible.

While it is not written for performance, the parser is efficient, thanks to the fact that data are not always copied.

It does not contain any unsafe code (*i.e* code where the compiler cannot provide static guarantees about behavior, like syscalls). Thanks to that, it should be immune to any kind of memory-related problems.

The parser does not decrypt the session nor perform cryptographic operations.

c) State machine: The state machine of TLS is known to be complex, and very often wrongly implemented in both client and servers [28]. Representing the state machine and following it after each received message is interesting for security, as it allows to detect violations caused by valid messages, received in the wrong order.

One could argue that if the state machine is very hard to implement, the same kinds of errors could happen in the TLS parser. While it is true, the consequences are very light: it would only result in a state detected as invalid, that is a false positive.

Two different methods have been tested: storing the state machine as a graph and make transitions depending on message types, or list all transitions and return the new states using pattern matching. The latter has been retained, because pattern matching offers a very elegant solution to match the message type and its content using the same syntax, and results in a compact and readable implementation: the entire state machine, including cases like not presenting a server certificate or anonymous Diffie-Hellman negotiation, is less than 100 lines.

The implementation is a function taking the previous state and the received message, and returning the new state, or an error for an invalid transition, for all unknown cases.

```
fn tls_state_transition_handshake(state: TlsState, msg: &
     TlsMessageHandshake) -> Result<TlsState,
     StateChangeError> {
    match (state,msg) {
        (None,        &ClientHello(ref msg)) => {
            match msg.session_id {
                Some(_) => Ok(AskResumeSession),
                _       => Ok(ClientHello)
            }
        },
        // Server certificate
        (ClientHello, &ServerHello(_))
                        => Ok(ServerHello),
        (ServerHello, &Certificate(_))
                        => Ok(Certificate),
        // Server certificate, no client certificate
             requested
        (Certificate, &ServerKeyExchange(_))
                        => Ok(ServerKeyExchange),
        // [...]
        // All other transitions are considered invalid
        _ => Err(InvalidTransition),
```

Listing 11. TLS State Machine extract (simplified)

B. *Rust and Suricata: Rusticata!*

The parser described in the previous section is written in pure Rust. As explained earlier, the objective is not to rewrite Suricata, but to modify it to embed the safe parser. In this section, we explain the design of the proposed integration, and how C and Rust can be merged in a single software.

The Rusticata project is an implementation of fast and secure parsers in Rust, along with a simple abstraction layer to use them in Suricata. Rusticata has been published with an open-source license [1].The project itself is divided in several parts:

- Parsers written using Rust and nom
- Suricata, extended with an application layer
- A glue layer between both, to propose a generic and safe API

To minimize the number of modifications required in Suricata, the Rust code is compiled as a shared library exposing C functions. This very simple design allows to write the C code in exactly the same way as if the library contained C code. The complete architecture, including components and function calls, is depicted in figure 2.

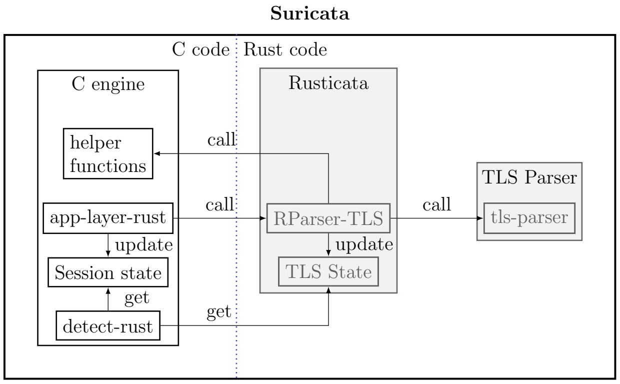

Figure 2. Function calls between C and Rust.

The main program (configuration, event loop, data acquisition and processing, etc.) remains the same.

Only one specific part is added: a fake application layer. In Suricata, an *application layer* is a set of functions registered to the engine to parse a specific application protocol. These functions are called when a packet matching the protocol is received, and maintain a state (a set of stored variables) for each session.

The added application layer is a simple proxy, which declares the parser for the TLS protocol, and forwards all calls to the Rust library. This design is modular and allows including more parsers when they are ready.

1) Exchanging data between Rust and C: In Rust, simple functions (non-generic) can be exported to use the C ABI using the `extern` specifier, coupled with the `#[no_mangle]` attribute to prevent mangling function and variable names. Using this attribute, the marked Rust functions are seen as regular C functions and are called directly from the C engine, as shown in figure 2.

While this allows a very easy way to call functions, this leaves a problem for data: except some primitive types, the memory model is not the same in the two languages.

Rust offers an attribute `#[repr(C)]` for structures, which tells the compiler to use the same representation the C compiler would use for this object. This provides a very convenient

[1] https://github.com/rusticata

way to access Rust objects from the C code, without any additional costs for data conversion or access.

However, this is not the implemented solution in Rusticata. Indeed, it does not help having a clean design of the code, because while inter-languages access are efficient, it brings back the memory problems described in this article, for example if one side accesses an object freed by the other.

So while the objects can be accessed directly, in the Rusticata project Rust structures are stored in C as opaque pointers, and accessors functions have been added. The additional cost is balanced by the safety of the memory management, and the fact that each language only cares for the objects allocated in its own code.

2) Using the project main functions: A very important point, when trying to propose a different language for integration, is to avoid recoding or duplicating other parts of the code like support functions. For example, logging functions, code to send alerts, signatures detection code, etc.

This brings the need to be able to use the C functions from Rust, which is the opposite of the previous data exchange. To do that, the Rust code must know which C functions to call. For example, during the initialization of the Rust library, it looks for the C functions and stores them into safe wrappers, in a Rust API. This allows the Rust code to call the logging functions using the standard `log` crate, but sending log messages through the base program log functions.

Using this method, Rusticata uses all support functions like the log and alert functions, and the signature engine from Suricata.

3) Thread safety: Suricata uses a lot of threads to handle multiple sessions concurrently, so the parser functions must be thread-safe. It is also important to note that the threads are created in the C side, so the Rust functions are not aware of these threads.

The Rust ownership system that prevents memory errors also helps preventing data races. Every data type knows whether it can safely be sent between or accessed by multiple threads, and Rust enforces this safe usage; there are no data races, even for lock-free data structures [29]. The compiler checks every variable and type: for example, mutable global variables are forbidden (they must be wrapped in either a locking or a guarded structure), and data types that cannot introduce memory unsafety in threaded programs must implement the `Send` or `Sync` traits.

To preserve these properties when wrapping C code, this means that all `unsafe` code must be treated carefully, but the native Rust code is guaranteed to be thread-safe. To do that, we limit the use of `unsafe` to the only operations of reading and writing buffers from C (these buffers are allocated per-thread by Suricata) and write all parsing code in native Rust. The parsing functions do not store any context and are all reentrant, to make easier to check for thread-safety.

C. Performance and Results

1) Functionalities: Before replacing the C parser, the new one must provide at least the same functionalities. The Rust TLS parser does in fact much more than the C version, because it is able to deeply parse the structures, while the C version only read a few fields, and for example is not able to decode the TLS extensions. This allows to write more elaborate rules for the IDS, like detecting the use of small Diffie-Hellman parameters.

In order to check the functionalities, it is possible to write tests based on the existing test vectors and expected results. Suricata comes with many unit tests, which helps ensuring that the features not only are similar, but also behave the same way as in the previous code.

2) Benchmarks: A few benchmarks were realized for the different parts of the TLS parser. It was not very relevant to compare it with the current TLS parser written in C in Suricata, as it has less features. In our tests, the speeds were still very similar. The tests were realized on a machine running a single Core i5 with 2 cores with hyperthreading. The tests measure the average, minimum and maximum duration of the parsing functions.

We can observe in the Rust parser a very consistent behavior, and the ability to parse a complete handshake in a time in the order of a millisecond.

```
bench_tls_rec_certificate  ... bench:  247 ns/iter (+/- 54)
bench_tls_rec_clienthello  ... bench:  579 ns/iter (+/- 102)
bench_tls_rec_serverdone   ... bench:  146 ns/iter (+/- 18)
bench_tls_rec_serverhello  ... bench:  193 ns/iter (+/- 10)
bench_tls_rec_serverkeye.. ... bench:  145 ns/iter (+/- 1)
```

The way the Rust compiler works gives a few benefits out of the box. It tries to apply static dispatch as much as possible, so some high level constructs like closures have little to no cost. The assembly generated is very close to the one a C compiler would write, as seen in the appendix. There's a conscious effort from the community to make memory handling easy to vectorize, and to remove runtime bound checks and null checks when the compiler can guarantee it is safe.

Additionally, nom uses almost no heap allocation and stores as much as possible on the stack. This makes it easy to bound the memory usage and avoid the unpredictable delays of memory allocators. The code generated by nom is very linear and easy to analyze for a compiler. It mainly contains branches based on the result of a previously executed parser, and no loop or jumps based on the currently observed byte. That kind of code can be merged in linear blocks and makes branch prediction easier.

3) Fuzzing: Fuzzing tools such as AFL [30] try to assess the security of a program by sending random content, and detect incorrect behavior. We have used AFL-rust [31] a variant of AFL modified for Rust programs. This tool allows to apply fuzzing for functions input, and not only the program input. It also tracks the execution path inside functions, to detect if a new path is discovered, or if all paths inside a function are explored. This is particularly interesting to target, for example, functions containing unsafe code, but since there is no such function in the TLS parser, the most interesting functions are the functions handling the record and message fragmentation, and parsing the content in depth.

We selected the top-level parsing functions (TLS record and messages) of the TLS parser, providing 4 different initial vectors: `ClientHello`, `ServerHello`, `Certificate`, and `ServerKeyExchange`. The fuzzing was realized using 5 parallel processes on a dual Xeon CPU, each having 6 cores with hyperthreading, during 20 days, during which the tests were executed 4054 million times. AFL-rust was stopped after a long period where no new path was discovered. In our case, no new path was discovered after the first 12 days.

There was no crash during that time, and very interestingly, no hang (program not responding after a timeout, for example because of an infinite loop). These results tend to confirm the expected security properties of the parsers. However the authors would like to insist on the fact that this is not a proof of security. The results are however very interesting when considering the fact that the parser was developed very quickly, and did not require a long time for stabilization and debugging.

VII. Conclusion

In this paper, we propose a pragmatic solution to the vulnerabilities of C programs manipulating untrusted data. The solution is a two-step approach: first, using Rust (or a language providing similar properties) to prevent memory-related bugs, and using a parser combinator to improve the implementation correctness and prevent missing tests or logical bugs. We have experimented this solution on two large, widely used programs, and tested the efficiency and robustness of the results.

The development of the parsers was a good opportunity to test our claims on the language. The main benefit is very simple yet representative of using a memory safe language: no debugger was used during the development. Similarly to a classical claim for functional languages, the compiler checks result in a slightly longer write-compile iterative process, but regaining much more time thanks to the fact the debugger is not needed. In particular, pattern matching plus strong typing is a very powerful combination to enforce good coding rules, and do not leave any possible execution branch untested. The fact that the compiler enforces these checks is a key point of the solution: it provides a systemic and global approach of memory safety, and ensures no test is missing. It also provides global thread safety, statically checked by the compiler.

In addition to the memory safety, one of the main benefits is that the time required for development and testing was greatly reduced. Several other parsers than TLS were added afterwards. Once the language and the parser generator were well understood, adding a parser for a new protocol is just a matter of a few hours.

One very interesting property of the generated parsers is that they tend to use only the stack, and not allocate any memory. This is good to control memory pressure, but also makes it possible to use very restrictive operating system mechanisms like the strict mode of `seccomp`: this mode allows only 4 system calls and forbids memory allocations.

Are all the possible bugs solved ? No, some remains like logic or algorithmic errors. But at least, the generated parsers and the application using them will not crash. All possible memory corruptions are also not prevented: they are only prevented in the Rust code, under the hypothesis that it is called properly from the C code. A possible solution is to progressively extend the part of Rust code by rewriting other components and eliminate the *unsafe* code. This is possible thanks to the pragmatic, step-by-step approach, but advocates for a nearly-full rewrite, which is a different task.

There are a few drawbacks. First, learning a new language is not easy, and developers must be convinced that the benefits overcome the cost to learn it. This is not always the case, and should be well considered before starting a project [32], or efforts will be wasted. A common comment on the Rust language is that it suffers from a bad readability. This is questionable, and in fact it is often the sign of the lack of habit which quickly disappears. It also takes time to go further than just using a new language and get used to its code patterns and best coding practices.

Proposing a parser in a new language also means that convincing the developer community will be a challenge. Maintaining software written in two languages is an effort, the decision to switch is very important. The community must be convinced that the effort required is worthy, and that the new parsers have the same functionalities. Using the unit tests from the project (when they exist) is a good way to prove it.

Switching from C to Rust also cause problems with strong typing: it is hard for some developers to come from a very permissive language, and have to write a clean program with respect to types and lifetimes of objects or return values. To the authors, it is not a drawback, but a good point: it is harder at the first time, but it really improves code quality and security.

Rust and nom allow writing parsers quickly while being safe. After the initial learning phase, the time required to add a new parser is quite small, and can become a convincing argument.

Developing safe parsers should now be encouraged in a larger community. We hope that providing good implementations in Rust can be used to write new thin abstraction layers in other software, for example Wireshark, to improve their security.

Appendix
Verifications on compiled code

This section describes how some of the properties of Rust are translated in compiled code, to analyze whether some patterns will be efficiently or securely implemented at runtime. Rust is based on the LLVM toolchain, so the verifications were done by compiling Rust code to LLVM intermediate runtime language, and analyzing it.

a) Memory model: Rust uses compact structures as much as possible, and adds very few overhead compared to C. For example, the following code:

```
struct Foo<’a> {
    a: u8,
    b: u32,
    c: &’a[u8],
}
```

is translated to the following LLVM declaration:

```
%Foo = type { i8, i32, { i8*, i64 } }
```

The resulting structure is the same as a C declaration would use. We also note that a *slice* is implemented as a pointer with a size.

b) Zero-copy: As parsers manipulate a lot of buffers and slices, We want to verify if the parsing, and the exchange of structures containing slices does not imply copying the underlying data. Using the previous structure, we declare a simple parser using nom:

```
named!(parse_foo<Foo>,
    chain!(
        a: be_u8 ~
        b: be_u32 ~
        c: take!(a),
        || { Foo{a:a,b:b,c:c}}
    )
);
```

We then extract the LLVM code for this parser:

```
  %25 = bitcast %"12.nom::IResult<&[u8], Foo, u32>"* %foo
       to i8*
  %26 = getelementptr inbounds %"12.nom::IResult<&[u8], Foo
       , u32>", %"12.nom::IResult<&[u8], Foo, u32>"* %foo,
       i64 0, i32 0
  %27 = getelementptr inbounds %"12.nom::IResult<&[u8], Foo
       , u32>", %"12.nom::IResult<&[u8], Foo, u32>"* %foo,
       i64 0, i32 2, i64 0
  %28 = getelementptr inbounds %"12.nom::IResult<&[u8], Foo
       , u32>", %"12.nom::IResult<&[u8], Foo, u32>"* %foo,
       i64 0, i32 2, i64 1
  %29 = getelementptr inbounds %Foo, %Foo* %r, i64 0, i32 0

  %318 = ptrtoint i8* %293 to i64
  store i64 0, i64* %26, align 16, !alias.scope !113, !
       noalias !114
  store i64 %res.sroa.5.0.i, i64* %27, align 8, !alias.
       scope !113, !noalias !114
  store i64 %res.sroa.7.0.i, i64* %28, align 8, !alias.
       scope !113, !noalias !114
  store i8 %291, i8* %31, align 8, !alias.scope !113, !
       noalias !114
  store i32 %317, i32* %tmp70.sroa.4123.0..sroa_cast.i,
       align 4, !alias.scope !113, !noalias !114
  store i64 %318, i64* %tmp70.sroa.5.0..sroa_idx125.i,
       align 8, !alias.scope !113, !noalias !114
  store i64 %295, i64* %tmp70.sroa.6.0..sroa_idx127.i,
       align 8, !alias.scope !113, !noalias !114

  call void @llvm.lifetime.start(i64 24, i8* %29)
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %29, i8* %31,
       i64 24, i32 8, i1 false)
```

Here is an explanation of this code: the first instructions store the pointers corresponding to every field of the structure. The following instructions store the elements read from the stream and stores them into the structure. During the copy, we can check that a new slice is created, but only the address and length of data is copied.

Finally, the last line seems surprising (a `memcpy`), but this copy is used to copy the structure itself into the return value of the functions, it is not deep-copied.

c) Accessing a slice: Every access to an array or a *slice* is verified. For example, the following code:

```
let _ = foo.c[1295];
```

is compiled with guards added:

```
bb32:                   ; preds = %bb32.loopexit, %bb29
  %329 = icmp ugt i64 %320, 1295
  br i1 %329, label %bb34, label %panic, !prof !140
```

At runtime, the index is tested, and an invalid access results in killing the program :

```
thread 'main' panicked at 'index out of bounds: the len is
    1 but the index is 1295', src/main.rs:32
```

Obviously, stopping the program is not a very useful behavior, even if that makes at least the program non-exploitable. In practice, Rust encourages to use iterators instead of array indices, preventing the problem.

d) Integer overflows/underflows: Rust uses two different compilation modes: debug and release. In debug mode, arithmetic on signed and unsigned primitive integers is checked for overflow, and the program stops if it occurs. In release mode, overflow is *not* checked, and programs are vulnerable to integer overflows/underflows.

Even if they are a bit harder to exploit in Rust, since they cannot result in a buffer overflow and memory corruption, they can still have consequences since it can be used to change to control flow to use a wrong branch of a test, for example. However, the language offers a solution: using the `overflowing_<op>` family of operations, the LLVM instructions with proper verifications will be used. The only drawback is that since it is not automatic, the developer can miss some checks.

e) Non-executable memory: This feature, ensuring that the sections of the compiled objects are strictly writeable or executable, but not both, is active by default.

f) Randomization: ASLR is supported, and Rust code is compiled with the `-pie` flag by default.

g) Stack Protector: Unfortunately, Rust code has *no* protection activated for the stack. Admittedly, the generated code is supposed not to contain any overflow thanks to the compiler checks, but it is a bit sad that a well-established function integrated to LLVM is not used, even if the integration causes some difficulties (tracked in Rust issue #15179).

That also means that a Rust library loaded from a C executable will provide ROP gadgets. Hopefully, the library supports ASLR, but this is not satisfactory, and could be improved in the future.

h) Immediate resolution (bindnow): The `bindnow` compiler flag allows to resolve all dynamic symbols at start-up, instead of on-demand. It is not active by default, but the option can be activated by adding the following to the file `.cargo/config`:

```
[build]
rustflags = ["-C","link-args=-Wl,-z,relro,-z,now"]
```

References

[1] "CVE details," http://www.cvedetails.com, 2016.

[2] J. Drake, "Stagefright: Scary code in the heart of android," https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf, 2015, BlackHatUSA.

[3] N. Seriot, "Parsing JSON is a minefield," http://seriot.ch/parsing_json.html, 2016.

[4] Cloudflare, "Incident report on memory leak caused by cloudflare parser bug," https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/, 2017.

[5] P. Chifflier and A. Fontaine, "Architecture systme d'une sonde durcie," Conference C&ESAR, 2014.

[6] E. Jaeger, O. Levillain, and P. Chifflier, "Mind your language(s): A discussion about languages and security (long version)," https://www.ssi.gouv.fr/uploads/IMG/pdf/Mind_Your_Languages_-_version_longue.pdf, 2014.

[7] E. Jaeger and O. Levillain, "Mind your language(s): A discussion about languages and security," *2014 IEEE Security and Privacy Workshops (SPW)*, 2014.

[8] W. Sewell, "Golangs real-time GC in theory and practice," https://blog.pusher.com/golangs-real-time-gc-in-theory-and-practice/, 2016.

[9] J. Turner, "Dropboxs exodus from the amazon cloud," https://news.ycombinator.com/item?id=11283688, 2016.

[10] Joyent, "Joyent HTTP parser," https://github.com/nodejs/http-parser/blob/master/http_parser.c, 2009.

[11] C. Networks, "Ragel state machine compiler," http://www.colm.net/open-source/ragel/.

[12] G. Couprie, "Nom, a byte oriented, streaming, zero copy, parser combinators library in rust," *2015 IEEE Security and Privacy Workshops (SPW)*, 2015.

[13] H.-J. Boehm, R. Atkinson, and M. Plass, "Ropes: an alternative to strings," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9450&rep=rep1&type=pdf, 1995.

[14] G. Couprie, "nom benchmarks," https://github.com/Geal/nom_benchmarks, 2016.

[15] VideoLAN, "VLC media player," https://www.videolan.org/.

[16] VideoLAN, "VLC security advisories," https://www.videolan.org/security/.

[17] A. S. Incorporated, "Video file format specification version 10," https://www.adobe.com/content/dam/Adobe/en/devnet/flv/pdfs/video_file_format_spec_v10.pdf, 2008.

[18] G. Couprie, "flavors: a rust FLV parser," https://github.com/Geal/flavors.

[19] S. project, "rust-bindgen," https://github.com/servo/rust-bindgen?files=1, 2015.

[20] S. Marshallsay, "rusty-cheddar," https://github.com/Sean1708/rusty-cheddar, 2015.

[21] G. Couprie, "vlc-module.rs," https://github.com/Geal/vlc_module.rs, 2016.

[22] Open Information Security Foundation, "Suricata: Open source IDS/IPS/NSM engine," https://suricata-ids.org/.

[23] O. Levillain, "SSL/TLS, 3 ans plus tard." SSTIC, 2015.

[24] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 223–238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/kaloper-mersinjak

[25] "CVE-2014-0160." Available from MITRE, CVE-ID CVE-2014-0160., Dec. 3 2013. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[26] T. Pornin, "BearSSL: Fixed buffer overflow and NULL pointer dereference," https://bearssl.org/gitweb/?p=BearSSL;a=commit;h=e8ccee8bcdae80cdf74c6d7327f1c7572589fae3, 2016.

[27] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud *et al.*, "A messy state of the union: taming the composite state machines of TLS," in *IEEE Symposium on Security & Privacy 2015 (Oakland'15)*, 2015.

[28] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[29] A. Turon, "Fearless concurrency with Rust," https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html, 2015.

[30] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[31] C. Richardson, "afl.rs," https://github.com/frewsxcv/afl.rs.

[32] J. Sharp, "Which projects should convert to Rust?" http://jamey.thesharps.us/2017/01/which-projects-should-convert-to-rust.html, 2017.