Verified Parser Generation for Security Critical Applications

Nik Swamy
OPLSS 20201
https://project-everest.github.io/everparse/

Tahina Ramananandro, Aseem Rastogi, Tej Chajed, Antoine Delignat-Lavaud, Cedric Fournet, Nadim Kobeissi, Guido Martinez, Jonathan Protzenko, Irina Spiridonova
Incorrect handling of attacker-controlled inputs ➔ Leading cause of software security attacks

### 2020 CWE Top 25 Most Dangerous Software Weaknesses

#### Introduction

The 2020 Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses (CWE Top 25) is a demonstrative list of the most common and impactful weaknesses from the previous two calendar years. These weaknesses are dangerous because they are often easy to find, exploit, and can allow adversaries to completely take over a system from working. The CWE Top 25 is a valuable community resource that can help developers, testers, and users — as well as project managers, security researchers, and others — provide insight into the most severe and current security weaknesses.

To create the 2020 list, the CWE Team leveraged Common Vulnerabilities and Exposures (CVE®) data found within the National Institute of Standards and Technology (NIST) Database (NVD), as well as the Common Vulnerability Scoring System (CVSS®) scores associated with each CVE. A formula was applied to the data to score each weakness and severity.

#### The CWE Top 25

Below is a brief listing of the weaknesses in the 2020 CWE Top 25, including the overall score of each.

<table>
<thead>
<tr>
<th>Rank</th>
<th>ID</th>
<th>Name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
<td>46.82</td>
</tr>
<tr>
<td>[3]</td>
<td>CWE-20</td>
<td>Improper Input Validation</td>
<td>33.47</td>
</tr>
</tbody>
</table>
Incorrect handling of attacker-controlled inputs

Dire in low-level code where input validation and parsing code is

- Hand written in C/C++
  - for performance
  - for deployability (e.g., in kernel)
  - for legacy

And errors are catastrophic due to a lack of memory safety
Incorrect handling of attacker-controlled inputs

In many guises, across the stack

Remote adversary

Untrusted Network

Well-crafted packet

Also deep within critical systems when traversing trust boundaries
Incorrect handling of attacker-controlled inputs

In many guises, across the stack

1. Hand-rolled data-exchange formats, hand-rolled parsers
   • What could possibly go wrong?

2. Standardized formats, hand-rolled parsers
   • Windows 10 Bad Neighbor: TCP/IP ICMPv6 Router Advertisement
     Improper parsing of variable length inputs leading to remote code execution/BSOD
   • Heartbleed:
     Improper parsing of variable length input leading to information disclosure

3. Standardized formats: buggy formats, buggy parsers
   • E.g., Malleability: leading to crypto vulnerabilities
     • PKCS #1 signature forgery, Bitcoin transaction malleability
Preventative Measures

It is our policy that any time a security problem is found, we will not only fix the problem, but also implement new measures to prevent the class of problems from occurring again. To that end, here’s what we’re doing to avoid problems like these in the future:

1. A fuzz test of each pointer type has been added to the standard unit test suite.
2. We will additionally add fuzz testing with American Fuzzy Lop to our extended test suite.
3. In parallel, we will extend our use of template metaprogramming for compile-time unit analysis (kj::Quantity in kj/units.h) to also cover overflow detection (by tracking the maximum size of an integer value across arithmetic expressions and raising an error when it overflows). More on this below.
4. We will continue to require that all tests (including the new fuzz test) run cleanly under Valgrind before each release.
5. We will commission a professional security review before any 1.0 release. Until that time, we continue to recommend against using Cap’n Proto to interpret data from potentially-malicious sources.

I am pleased to report that measures 1, 2, and 3 all detected both integer overflow/underflow problems, and AFL additionally detected the CPU amplification problem.

Integer overflow bugs

As the installation page has always stated, I do not yet recommend using Cap’n Proto’s C++ library for handling possibly-malicious input, and will not recommend it until it undergoes a formal security review. That said, security is obviously a high priority for the project. The security of Cap’n
Plus, many legacy formats remain designed for:

- Compactness
- ABI compatibility
- mmap’able

Serialization: `memcpy`

Parsing: `reinterpret_cast<T> . validate`
Standardized formats have their challenges too

Wire formats prescribed by RFCs in a semi-formal notation

Or in other notations like ASN.1

Are the formats well-designed?
  ◦ E.g., non-malleable?

Are their parsers and serializers correctly implemented?


```c
uint16 ProtocolVersion; opaque Random[32]; uint8 CipherSuite[2];

struct {
  ProtocolVersion legacy_version = 0x0303;
  Random random;
  opaque legacy_session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  opaque legacy_compression_methods<1..2^8-1>;
  Extension extensions<8..2^16-1>;
} ClientHello;
```
A Mathematically Proven Low-level Parser Generator

For a variety of formats, ranging from mmap’able binary wire formats to semi-formal RFC specs

Our goal

• Abolish writing low-level binary format parsers by hand
• Instead, specify formats in a high-level declarative notation
• Auto-generate performant low-level code to parse binary messages
• Integrate seamlessly with existing codebases in a variety of languages (C, C++, Rust, ...)

With formal proofs that:

• Formats enjoy various good properties, e.g., non-malleability
• Generated code is
  • Memory safe (no access out of bounds, no use after free etc.)
  • Arithmetically safe (no overflow/underflow)
  • Functionally correct (that it parses exactly those messages that conform to the high-level spec)
  • Free from double-fetches, so safe against time-of-check/time-of-use bugs

https://project-everest.github.io/everparse/
Starting from a high-level language of message formats

EverParse auto-generates parsing code that is

- Safe
- Correct
- Fast (zero-copy)

Correctness:
\[
\text{parse (serialize msg) = msg} \\
\text{valid msg => serialize (parse msg) = msg}
\]

Performance:
- ASN.1
- EverParse
- F* (type theory-based proof assistant and programming language)

A type-theory-based proof assistant and programming language
https://fstar-lang.org
### Hardening critical applications in C/C++

Since spring 2020

**Every network packet passing through Microsoft Hyper-V** is validated by EverParse

- Hyper-V: Core isolation technology of the Microsoft Azure cloud
- Multiple layers of headers, many verified already, further layers in progress

Custom binary wire formats designed to also be ABI-compatible and mmap’able

<table>
<thead>
<tr>
<th><em><em>Verified parsers and serializers for non-malleable wire formats in verified F</em> applications</em>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS 1.3 record and handshake messages</td>
</tr>
<tr>
<td>QUIC record layer messages</td>
</tr>
<tr>
<td>DICE Secure Measured Boot for IoT devices, ASN.1</td>
</tr>
<tr>
<td>Bitcoin transaction log validation</td>
</tr>
</tbody>
</table>

**All producing verified high-performance C code**
Hardening critical applications in C/C++

A diversity of existing wire formats
  - Designed for efficiency, compactness and ABI compatibility
  - So that parsing and serialization can be done by `memcpy/reinterpret_cast`

We designed a new specification language to capture a wide variety of these formats

Produce functionally correct, memory safe, double-fetch free validators of these formats in C

And interpose our validators at the attack surface to ensure that ill-formed data doesn’t reach the rest of the system
Dependent Data Descriptions in 3D:  
A source language of message formats

Augmenting C data types with constraints, variable-length structures, and actions

Refinement types for data validity constraints

typedef struct _SAMPLE {
    UINT32   MajorVersion { MajorVersion = 1 };
    UINT32   MinorVersion { MinorVersion = 0 };
    UINT32   Min;
    UINT32   Max { Min <= Max }
} SAMPLE;
Augmenting C data types with **constraints**, **variable-length** structures, and **actions**

**Contextually tagged unions with casetype**

```c
typedef union _MessageUnion {
    Init init;
    Query query;
    Halt halt;
} MessageUnion;

typedef struct _Message {
    UINT32 tag;
    MessageUnion message;
} Message;
```
Augmenting C data types with constraints, variable-length structures, and actions

Contextually tagged unions with casetype

castetype _MessageUnion(UINT32 tag) {
    switch(tag) {
    case INIT_MSG:
        Init   init;
    case QUERY_MSG:
        Query  query;
    case HALT_MSG:
        Halt   halt;
    }
} MessageUnion;

typedef struct _Message {
    UINT32 tag;
    MessageUnion(tag) message;
} Message;
Augmenting C data types with constraints, variable-length structures, and actions

Structures with variable-length fields

typedef struct _VLDATA {
    UINT32 ByteLength;
    SAMPLE Samples[:byte-size ByteLength]
} VLDATA;
Augmenting C data types with **constraints**, **variable-length** structures, and **actions**

**Structures with variable-length fields**

typedef struct _VLDATA(UINT32 TotalMessageLength) {
    UINT32 ByteLength;
    UINT32 Offset
    { is_range_okay(TotalMessageLength, Offset, ByteLength) &&
      Offset >= sizeof(this) };
    UINT8 Padding[:byte-size Offset - sizeof(this)]
    SAMPLE Samples[:byte-size ByteLength]
} VLDATA;
Augmenting C data types with constraints, variable-length structures, and actions

Imperative actions for selective parsing and further validation

typedef struct _Sample(mutable UINT32 out) {
    UINT32   MajorVersion { MajorVersion = 1 };
    UINT32   MinorVersion { MinorVersion = 0 };
    UINT32   Min;
    UINT32   Max { Min <= Max }
        { :on-success *out = Max}
} SAMPLE;
Augmenting C data types with constraints, variable-length structures, and actions

Imperative actions for selective parsing and further validation

typedef struct _A(mutable UINT32* accum) {
   ...
   UINT32 NumBEntries
   {on-success accum += NumBEntries }
} A;

typedef struct _B(mutable UINT32* expectedB)
{ ... {on-success expectedB--} } B;

typedef struct _C(UINT32 TotalLength,
   mutable UINT32* accum) {
   UINT32 SizeOfAs;
   A(accum) As[:byte-size SizeOfAs];
   B(accum) Bs[:byte-size TotalLength -SizeOfAs - 4]
   {on-success return (*accum == 0) }
} C;
Generated C code, after verification

- C code aims to be human-readable, human patchable
- Propagates comments from source spec
- Generates predictable descriptive names

Theorems:
- CheckPacket returns true if and only if the bytes in *base contains a valid representation of the format specification for a Packet
- CheckPacket reads no byte of *base more than once
- Mutates at most the out parameters, dataOffset … perPacketInfoLength in a type-correct manner

```
BOOLEAN
CheckPacket(
    uint32_t ___PacketLength,
    uint32_t ___HeaderLength,
    uint32_t *dataOffset,
    uint32_t *dataLength,
    uint32_t *perPacketInfoOffset,
    uint32_t *perPacketInfoLength,
    uint8_t *base,
    uint32_t len);
```

Insert a call to CheckPacket on attack surface
Experience, spec archaeology, ...

• Developed specifications for 4 core message formats for various virtualized device
  • Ultra-high value scenarios: Security bugs here are catastrophic for the entire cloud
  • Layered protocols, with incremental parsing
  • Many more to come

• Totaling around 6000 lines of specification in 3d
  • Automatically generating ~30KLOC of verified C code
  • Working on a clang-based frontend to better integrate 3d specs with C headers

• Highly performance sensitive in certain scenarios
  • Target: Less than 2% measured performance overhead
  • Result: Overhead is unmeasurable
    • In some cases, our code is more efficient, since we can aggressively avoid copies that were previously incurred due to defenses against double fetches

• Main challenge: Discovering a specification
  • Proprietary specs, intentionally divergent from official standards from which they were derived
  • Backward compatibility & complex testing matrices
Using EverParse with Verified F* Applications

**TLS (miTLS)**
- Verified TLS secure channel with formal security model (IEEE S&P 2017)
- Memory-safe, functionally correct, secure
- Handshake verification in progress
- USENIX Security 2019: TLS handshake message formats

**QUIC (EverQUIC)**
- Verified QUIC record layer with formal security model
- Memory-safe, functionally correct, side-channel resistant, secure

**DICE/RIoT (DICE*)**
- Verified measured boot for embedded devices (secured boot with measurements)
- Memory-safe, functionally correct, side-channel resistant
- Submitted: ASN.1 X.509 certificates
Performance Results

<table>
<thead>
<tr>
<th></th>
<th>QD</th>
<th>F* LoC</th>
<th>Verify</th>
<th>Extract</th>
<th>C LoC</th>
<th>Obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS</td>
<td>1601</td>
<td>70k</td>
<td>46m</td>
<td>25m</td>
<td>190k</td>
<td>717KB</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>31</td>
<td>2k</td>
<td>2m</td>
<td>2m</td>
<td>2k</td>
<td>8KB</td>
</tr>
<tr>
<td>PKCS1</td>
<td>117</td>
<td>5k</td>
<td>3m</td>
<td>3m</td>
<td>4k</td>
<td>26KB</td>
</tr>
<tr>
<td>LowParse</td>
<td>33k</td>
<td>4m</td>
<td>2m</td>
<td>0.2k</td>
<td>1KB</td>
<td></td>
</tr>
</tbody>
</table>

Takeaway:
• Scales to large data formats
• Code produced is fast
Parser and Serializer Specifications

let parser (t: Type) = (p: bytes → option (t × ℤ)) {
∀ (b1 b2: bytes). match p b1, p b2 with
| Some (x1, len1), Some (x2, len2) ->
  x1 = x2 ==> slice b1 0 len1 = slice b2 0 len2
| _ -> True
}

let serializer (#t: Type) (p: parser t) = (f: t → bytes) {
∀ (x: t). p (f x) = Some (x, length (f x))
}

With parser refinement:
• Injectivity

And more:
• Consumption bounds
• Strong prefix property
• Etc.
Controlled by metadata
Validators

type validator (#t: Type) (p: parser t) =
(b: bytes) →
(res: bool {
    res = true ⇔ Some? (p b)
})

Not compositional!
Validators

```
type validator (#t: Type) (p: parser t) =
  (b: bytes) →
  (res: option nat { 
    match p b with
    | None → res = None
    | Some (_, consumed) → res = Some consumed
  })
```

What are bytes in C?
Validators

type validator (#t: Type) (p: parser t) =
  (b: buffer UInt8.t) →
  (len: UInt32.t { len = length b }) →
  ST (option UInt32.t)
  (requires λ mem → live mem b)
  (ensures λ mem res mem' →
    modifies loc_none mem mem' /
    match p (as_seq mem b), res with
    | None, None → True
    | Some (_, consumed), Some consumed' → consumed' = consumed
    | _ → False
  )
**Validators**

\[
\text{type validator } (\#t: \text{Type}) (p: \text{parser } t) = \\
(b: \text{buffer } \text{UInt8.t}) \rightarrow \\
(\text{len: UInt32.t } \{ \text{len }= \text{length } b \}) \rightarrow \\
\text{ST (option UInt32.t)} \\
(\text{requires } \lambda \text{mem }\rightarrow \text{live mem b}) \\
(\text{ensures } \lambda \text{mem res mem’ }\rightarrow \\
\text{modifies loc_none mem mem’ }\land \\
\text{match } p (\text{as_seq mem b}), \text{res with} \\
| \text{None, None }\rightarrow \text{True} \\
| \text{Some (_, consumed), Some consumed’ }\rightarrow \text{consumed’ }= \text{consumed} \\
| _ \rightarrow \text{False} \\
)\]
type validator (\textit{\texttt{Type}}) \ (p:\ \textit{parser}) \ (b:\ \textit{buffer Uint8.t}) \rightarrow
\ (\textit{len}::\textit{UInt32.t} \{ \textit{len} = \text{length} \ b \}) \rightarrow
\ ST \ (\textit{option Uint32.t})
\ (\textit{requires} \ \lambda \ \textit{mem} \rightarrow \ \text{live mem} \ b)
\ (\textit{ensures} \ \lambda \ \textit{mem} \ \textit{res} \ \textit{mem'} \rightarrow
\ \text{modifies loc\_none mem mem'} \ \backslash
\ \text{match} \ p \ \text{(as\_seq mem b)}, \ \text{res} \ \text{with}
\ | \ \text{None, None} \rightarrow \ True
\ | \ \text{Some} \ (_, \ \text{consumed}), \ \text{Some} \ \text{consumed'} \rightarrow
\ | \ _ \rightarrow \ False
)
type employee = {
  name : employee_name;
  salary : UInt16.t;
}

let employee'_parser : parser employee' =
  parse_pair employee_name_parser uint16_parser

let rewrite_employee (x : employee') : employee =
  let (name, salary) = x
  in
  { name = name; salary = salary; }

let employee_parser : parser clientHello =
  parse_rewrite employee'_parser rewrite_employee
let employee'_validator : validator employee'_parser =
 validate_pair employee_name_validator uint16_validator

let employee_validator : validator employee_parser =
 validate_rewrite employee'_validator rewrite_employee

type employee' = (employee_name × Uint16.t)

let employee'_parser : parser employee' =
 parse_pair employee_name_parser uint16_parser

let rewrite_employee (x: employee') : employee =
 let (name, salary) = x in
  { name = name; salary = salary; }

let employee_parser : parser employee =
 parse_rewrite employee'_parser rewrite_employee
type slice = {
  base: buffer UInt8.t;
  len: UInt32.t { len ≤ length b / \ len ≤ max_length };
}

let employee'_validator : validator employee_parser =
  validate_pair employee_name_validator uint16_validator

let employee_validator : validator clientHello_parser =
  validate_rewrite employee'_validator rewrite_employee

typedef struct {
  uint8_t *base;
  uint32_t len;
} LowParse_Slice_slice;

uint32_t Employee_employee_validator(LowParse_Slice_slice input, uint32_t pos)
{
  uint32_t pos1 = Employee_name_employee_name_validator(input, pos);
  if (pos1 > LOWPARSE_LOW_BASE_VALIDATOR_MAX_LENGTH)
    return pos1;
  else if (input.len - pos1 < (uint32_t)2U)
    return LOWPARSE_LOW_BASE_VALIDATOR_ERROR_NOT_ENOUGH_DATA;
  else
    return pos1 + (uint32_t)2U;

validate_pair and validate_rewrite inlined

uint16_validator inlined
Bang for the buck: Focus on parsing, protect the attack surface