Best Practices for Secure Data Intake, Data Modeling, and Data Design

Learnings from the Safe Documents (SafeDocs) Program

Sergey Bratus

Defense Advanced Research Projects Agency/Information Innovation Office (DARPA/I2O)

Executive summary

Design and implementation errors in software that receives and validates electronic data lead to vulnerabilities that may lead to mission failure. To reduce risk to the mission, electronic data formats MUST be modeled down to the wire format level, and the software code that ingests and validates data SHOULD be automatically generated from such models.

Background

Data objects. Sensor Open Systems Architecture (SOSA) software and hardware components send and receive data in the form of electronic messages and streams. These messages consist of data objects from the SOSA data models and of auxiliary data (a.k.a. metadata[1]) used to encode these objects into strings of bits and bytes (e.g., for on the wire transmission or storage), and to authenticate and configure components and connections. Examples of auxiliary data include codes for types and subtypes of messages, lengths of variable-length fields of data structures representing objects, encoded cryptographic credentials such as keys, and so on.

Data relationships. Typically, certain relationships are expected to hold between the data objects themselves and between the data objects and the associated auxiliary data. For example, an object representing a relative time value must be preceded by another object that represents an absolute time value. As another example, the actual length of an encoded variable-length value must correspond to the element(s) that describe the lengths of that value (if any) and of objects enclosing that encoded value in their own encoding, the sum of the lengths of parts should be less or equal to the length of the whole, and so on. As a general rule, when two or more elements of auxiliary data affect the processing of the message, their values must agree so as not to result in conflicting interpretations of the same message.

[1] Although *metadata* is sometimes misunderstood to imply that it doesn't need to be formally modelled or checked, this is a dangerous misconception. At some point some code will interact with the supposedly passive metadata (even just attempting to display it)—and will get exploited by it! Log4Shell and various Unicode injection attacks forcefully demonstrate that "metadata" is still "data" and needs to undergo the same thorough validation as any other data.

Vulnerabilities of data intake. To be transmitted over the wire or air, data objects are converted to sequences of bits or bytes, i.e., *serialized* by the sending component. Receiving software interprets the serialized representation to reconstruct the data objects and must validate any relationships that may be assumed by subsequent software code processing these objects. Failure to validate expected relationships typically leads to vulnerabilities. For example, the code that attempts to (re)construct the absolute time stamp from a relative one may scan the message backwards for a preceding absolute time value and would fail if no such value is found, leaving the computation in an invalid state. When declared lengths of encoded fields disagree, e.g., an inner object is larger than the outer object, it attempts to allocate memory based on the outer object's supposed length and then fills that memory with the decoded content of the inner object which may lead to overwriting memory allocated to *other* objects, a.k.a. memory corruption, and so on. When two conflicting auxiliary data descriptions of an object's size or content are possible, different components of the system may end up unwittingly implementing these conflicting interpretations, leading to inconsistent global state of the system. All of these errors are known to have led to critical vulnerabilities, including the infamous Heartbleed.

Best practices

The code that reconstructs and validates incoming data should be generated from a machine-readable (a.k.a. mechanized) model of the data that declaratively describes both the structure of objects and their relationships and extends to the wire-level encoding of these objects and related auxiliary data.

Low-complexity, interoperable data format definitions and data models. Data format design should pay close attention to parsing complexity and interoperability while supporting a formal semantics. The data format must avoid vulnerabilities induced by complex parsing rules that compromise the earlier recommendation for strict validation of input data before any application processing. Interoperability across different implementations of the same format version and across different format versions is improved by ``virtuous intolerance'' (RFC 9413) where all deviations from the specification are treated as fatal. Data models must formalize the structure and invariants reflected in the data format definition.

Models must cover wire format. The mechanized model MUST refine the physical data model down to the wire format, i.e., to the bit-level representations of objects and metadata. This refinement is needed to preclude ambiguity of interpretation and resulting vulnerability-inducing implementation disagreements and to avoid data corruption errors while reconstructing the objects.

Models must cover data dependencies and relationships. The mechanized model MUST declare **all** relationships between different data objects, auxiliary data, and metadata[2]. This is necessary to ensure that no assumed but actually unvalidated data property is acted on by

[2] A relationship that is assumed but not checked will be acted on at some point by some code—and will likely help exploit that code. Also, the same caution about metadata as described in footnote 1 applies.

subsequent code, leading to inconsistent state, data corruption, or vulnerability. Since relationships in multi-value data formats are typically complex, enumerating their complete set is key to assuring that all are checked correctly by the implementation.

Data intake code should be auto-generated from models. The implementation code SHOULD be automatically generated from the mechanized model. The generator should produce the code that provably, correctly, and completely implements the structure and relationship validation checks. This is necessary for both complex protocols and messages, and for seemingly simple ones, as examples of vulnerabilities resulting from forgotten or mis-implemented checks abound for both kinds. Moreover, automatic generation of data structures from serialized input enables automatic consistency checks between sending and receiving applications to use models rather than code, which will considerably simplify these consistency checks and enable faster and more assured integration.

SOSA Alignment

The above best practices significantly improve the quality attributes of interoperability and securability of SOSA components.

Interoperability: Mechanized data models assure equivalent interpretation of all messages by all SOSA sensor components that deploy code automatically generated from these models. This avoids a common and often critical failure mode in which interpretations of the same data diverge among components of the system.

Moreover, equivalent interpretation of complex messages by independently developed parsers tends to be among the hardest properties to test for and certify conformance. Deploying automatically generated code across components will reduce the effort needed for testing, certification, and integration.

Securability: Mechanized data models with unambiguous and complete descriptions of validity enable checking and filtering of incoming data for well-defined properties at well-defined interfaces or architectural insertion points. This system property is essential for securability of the overall system in multiple scenarios, including triage of vulnerabilities and response to changing operating conditions, wherein certain messages can be blocked due to threats.

Zero Trust: Mechanized data models are essential for assuring zero trust properties of components and systems since authentication---essential to zero trust---can only happen after incoming messages are deserialized and authentication credentials such as cryptographic tokens are extracted from them. Vulnerabilities in deserialization code thus, can be exploited before authentication can happen and the malicious message's contents or commands can be discarded as unauthenticated or unauthorized. This makes exploitation of these vulnerabilities, known as pre-authentication vulnerabilities, particularly devastating.

Pre-authentication vulnerabilities destroy zero trust properties. Automatically generating pre-authentication code from mechanized data models of authenticating data mitigates the risk of pre-authentication exploits.

In addition to pre-authentication vulnerabilities, there is another existential threat for zero trust: chain-of-trust parsing vulnerabilities. This class of vulnerability lurks wherever cryptographically signed data objects must be processed in a well-defined order ("chain") to extend trust to the document or message, and further validation is based on processing of previously interpreted data. Any deviations from the order of processing or any disagreements about which objects or spans of data are covered by previously checked cryptographic credentials destroy the chain and the trust. Unambiguously defining the order, the objects and the coverage mitigates this existential risk.

Acquisition: In addition to the technical alignment of the best practices with SOSA, there is a solid business case to be made. The SOSA Acquisition and Contracting Guide outlines the business case for SOSA and includes the following acquisition goals:

1. Reduced acquisition cycle time and overall life-cycle cost
2. Ability to insert cutting edge technology as it evolves
3. Commonality and reuse of components among systems
4. Increased ability to leverage commercial investment

Following the best practices above is in fact crucial for achieving these business and acquisition goals. In particular, mechanized data models have already been shown to aid in all four of these acquisition goals, and the supporting technologies developed in the SafeDocs program are already showing their promise for both existing and under-development SOSA data formats.

Supporting technologies

DARPA's SafeDocs program developed several Data Description Languages (DDLs) for creating and refining mechanized models of incoming data. These DDLs are accompanied by environments and tools to help write, "debug", and test models in DDL and to generate input handling and input filtering code from these models.

The common capability of these models is to describe the structure of data objects, their representations down to the wire format, and—crucially for validation and assurance—of their mutual dependencies and relationships. The DDLs vary in style and primitives of their modeling approaches, to accommodate a wide variety of developer experiences. The supporting tools also vary in the software dependencies they require.

For example, the *Hammer* toolkit is built around a standard C library and requires no additional dependencies beyond a typical C/C++ build chain. Hammer also offers bindings that allow it to be used from Python, Java, .Net, and other popular languages. The *DaeDaLus* and *Parsley* kits

provide stronger guarantees at the cost of requiring additional dependencies such as Haskell and the PVS proof and reasoning system. See table 1 in the appendix.

The following tools can help software developers and cybersecurity/privacy researchers improve their organization's security posture in handling electronic documents. These range in functionality and specificity for a variety of uses. Check each description and click on the tool's links for additional information.

Programmer resources for describing data formats and auto-generating parsing code

- **DaeDaLus**: Data description language for defining data formats and generating memory-safe parsers in a variety of languages.
- **Hammer:** Declarative secure parser/scanner construction kit in C, based on the parser combinator approach but requiring no additional dependencies beyond the Hammer C library.
  Parsley: Data description language that combines grammars and constraints in a modular way to capture data formats such as MAVLink, PDF and Executable and Linking Format (ELF).
- **PVS/PVS2C:** Interactive proof assistant with C code extraction for defining efficient, correct-by-construction parsers and independent proof-of-parse verifiers.

Tools for understanding document collections and format rules

- **File Observatory**: System to enable visualization, search, and discovery of complex file format patterns and data.
- **Format Analysis Workbench**: Platform for running and analyzing the output from any number of parsers dealing with a single file or streaming format. It is a workbench for developing and applying tools that aid in understanding the de facto formats which naturally emerge from open standards.
- **Dowker tools for statistical inference of file format behaviors**: Provides the ability to classify file behaviors against an arbitrary number of Boolean features to help developers focus into unusual or interesting behaviors. A demonstration video is available at https://www.youtube.com/watch?v=i3wl2jdIZv8.
- **PolyFile**: A utility to identify and map the semantic structure of files, including polyglots, chimeras, and the so-called "schizophrenic" files that appear differently to different software readers. It can be used in conjunction with its sister tool PolyTracker for Automated Lexical Annotation and Navigation of Parsers, a backronym devised solely for the purpose of collectively referring to the tools as The ALAN Parsers Project.

Tools to understand behavior of existing parser code

- **PolyTracker**: A general-purpose tool for efficiently performing data-flow and control-flow analysis of programs. It is a LLVM pass that instruments programs to track which bytes of an input file are operated on by which functions. It outputs a database containing the data-flow information, as well as a runtime trace. PolyTracker also

provides a Python library for interacting with and analyzing its output, as well as an interactive Python REPL.
- **Graphtage**: Command-line utility and underlying library for semantically comparing and merging tree-like structures, such as JSON, XML, HTML, YAML, Property List (Plist), and CSS files.

Resources for the Portable Document Format (PDF)

- **Arlington PDF Model**: Named after the city in which DARPA is located, provides an authoritative and comprehensive document object model for PDF, facilitating enhancements to security and driving new modalities in specification development.
- **Digital Corpora Project Corpus**: Approximately eight million real-world PDFs available for research, testing, and evaluation.
- **SPARCLUR**: A collection of various wrappers for extant PDF parsers and/or renderers along with accompanying tools for comparing and analyzing the outputs from these parsers.

Tools to secure Python's data format overwhelmingly used in Artificial Intelligence research

- **Fickling**: Decompiler, static analyzer, and bytecode rewriter for Python pickle object serializations.

SafeDocs also developed tools for integrating DDLs with VisualCode and Emacs editors, as well as with Visual Studio. SafeDocs developed visual debugging tools for data formats such as PolyFile and Format Analysis Workbench (FAW), and tracing tools for parsers, such as PolyTracker. See appendix table 2 for additional SafeDocs tool information.

Insufficiency and anti-patterns of current approaches

Experience shows that today's approaches are insufficient to eliminate vulnerabilities in code that handle data intake. This insufficiency is due to a combination of weaknesses in existing best practices and persistence of design and implementation anti-patterns that should be avoided.

Hand-coding of data intake code should be avoided. First and foremost, hand-coding of parsers, deserializers, and validators of untrusted input data—i.e., potentially crafted, manipulated, or malicious data—should be avoided. This is especially true for languages without memory safety guarantees such as C and C++, because any memory operation that involves values derived from untrusted input may result in memory corruption when based on an unchecked assumption about input. For example, allocating memory based on a size value computed from the inputs may result in allocating insufficient space for a subsequent series of writes to that memory, resulting in overwriting and corrupting unrelated data in neighboring memory. That corrupted data may then be accessed by other code that does not expect it to be corrupted, compounding the unintended execution effects. Even reading data based on values derived from input may result in a vulnerability: unrelated data may be read and then included in outgoing messages, as was the case with the infamous Heartbleed vulnerability.

Simply put, hand-written parsing code should be deprecated in high assurance systems.

"Input sanitization" is an anti-pattern for high assurance systems. The practice of scanning inputs for known problematic bit or byte sequences and automatically changing these inputs into forms presumed benign, is, in fact, an anti-pattern that should be avoided. This practice, known as input sanitization, may be occasionally effective against known attacks by disrupting their payloads or injection mechanisms, but is also known to be both bypassed by attackers or even leveraged to turn benign inputs into malicious ones. This is because validity of a complex input is typically the property of an entire message rather than a property of a few easily recognizable "pathogen" bytes that can be checked locally without regard for the larger message context.

Simply put, the "sanitization" practice of spot checks and "fixes" based on partial scanning of complex messages should be avoided in high assurance systems.

Intermixing data intake checks and data processing leads to errors. Many current implementations of complex protocols intermix input-checking and application logic, introduces confusion and errors. It is often not clear at most points in the code which assumptions about the data relationships are checked and which are yet to be checked, which results in acting on yet unchecked assumptions and exploitation.

In particular, specifications of application protocols tend to come with tables that specify data field sizes and expected value types in machine-readable forms, but use English natural language statements to describe the interdependencies and relationships between these fields. The expectation is that some of the initial input-handling code can be automatically generated from the tables, whereas the rest of the English-described relationships will be checked by the custom application logic. In short, checking of the more advanced relationships is postponed until the data contents extracted from inputs actually begin driving the application—i.e., when it is too late to defend the application logic.

Simply put, purely syntactic checking of data that postpones the checking of object dependencies and relationships until the data is acted upon and intermixes checking with the application logic is not sufficient for high assurance systems.

Finally, **without a complete and unambiguous machine-readable model of input data, automated checking of the manually produced code for errors is incomplete and inefficient**, even for seemingly simple wire formats. Although automatic analysis of code is desirable, it is either ineffective or inefficient without a mechanized model, because analyses of this code must guess which properties are desired and which are incidental to or emergent from an implementation. When expectations of object/value properties and relationships between objects and values are not mechanized, they cannot be checked exhaustively. An expectation that is acted upon but not actually checked is liable to introduce memory corruptions, overflows, and other unintended behaviors. Without a complete and unambiguous description

of expectations, automatic checking is reduced to either heuristically looking for known coding mistakes and misses more complex conditions.

Simply put, neither static (e.g., "linting") nor dynamic (e.g., "fuzzing") automated analyses are sufficient for high assurance systems and should not be used as a sole basis of trust in that code.

Contrast this with code automatically generated from declarative descriptions of desired and expected data format structure and relationships, where the desired properties are explicit at every point. Indeed, **non-declarative manually coded data validity logic should be replaced with declarative-based automatically generated code.**

Appendix

SafeDocs DDLs, their outputs and their dependencies

DDL	Output	Dependencies
DaeDaLus	Parsers in C++ or Haskell	GHC 9.4.5 and a Hsskell package manager (e.g., Cabal 3.6). Offline install available with no dependencies.
Hammer	Parsers in C, C++, Python, Ruby, Perl, Go, PHP, .NET	SCons
Parseley	An interpreter in OCaml	OCaml 4.11.1 or later and Opam 2.0 or later.

SafeDocs performer tools and integrations

SRI International

Tool Name	URL	Platform or Deps	License
Hashashin	https://github.com/riverloopsec/hashashin	python, BinaryNinja	MIT
Description	"Hashashin leverages IL based fuzzy hashing techniques to generate a compiler/architecture agnostic fingerprint which can be used for identifying the source of an unknown binary."		

Tool Name	URL	Platform or Deps	License
Data-iop Analyzer	N/A	eBPF, awk, strace	Proprietary
Description	The data-iop Analyzer is a part of Narf's PolyDocPoC tool that uses the `bpftrace` tool to derive a dynamic trace of the "data I/O protocol": the format-relevant I/O behavior of a PDF parser. It also includes a post-processing tool for strace-derived system call traces that models the state of major sequences of PDF operations, such as profiles of memory buffer use and gaps (unread bytes) in files."		

Tool Name	**URL**	**Platform or Deps**	**License**
PolyDocPoc & Sir-Parse-a-Lot	N/A	Polyswarm network, python, Docker	Proprietary
Description	These tools are parser instrumentation frameworks that apply a set of Analyzers to data from the PolySwarm threat intelligence artifact network to conduct format-based detection of malware in complex files.		

Tool Name	**URL**	**Platform or Deps**	**License**
MEOW suite	https://github.com/SRI-CSL/safedocs-meow	Intel Pin	MIT
Description	“MEOW is a tool for format-aware tracing of memory events based on Intel Pin”		

Tool Name	**URL**	**Platform or Deps**	**License**
NITF KaiTai Struct grammar	https://formats.kaitai.io/nitf/index.html	Minimal KaiTai Struct v0.8	MIT
Description	“NITF KaiTai Struct grammar defines an open source KaiTai spec utilized for parsing NITF files"		

Tool Name	**URL**	**Platform or Deps**	**License**
NITF Mutator	N/A	python	MIT
Description	“Library for manipulating NITF files used to evaluate NITF parsers”		

Tool Name	URL	Platform or Deps	License
SRI Recognizer	https://github.com/SRI-CSL/safedocs-recognizer	Go, python, Docker, Shell	MIT
Description	"DARPA SafeDocs software suite to bundle and orchestrate various format-aware tracing tools."		

Lockheed Martin Advanced Technology Laboratories

Tool Name	URL	Platform or Deps	License
parseLab	https://github.com/lmco/parselab	python	Apache v2.0
Description	parseLab is a modular framework to generate protocol parsers, fuzz protocol messages and provide capability for building custom protocol parser generators		

Trail of Bits

Tool Name	URL	Platform or Deps	License
PolyTracker	https://github.com/trailofbits/polytracker	Linux	Apache v2.0
Description	PolyTracker is a general purpose tool that adds instrumentation to programs at compile time to track all data flows through execution. It can map all input bytes to the program outputs they influence. PolyTracker's data flow analysis has been used to automatically detect parser bugs and differentials.		

Tool Name	URL	Platform or Deps	License
PolyFile	https://github.com/trailofbits/polyfile	Python 3 (cross-platform)	Apache v2.0
Description	PolyFile is a file format identification and exploration tool. It is a drop-in replacement for libmagic's file command and can also emit an interactive file explorer.		

Tool Name	**URL**	**Platform or Deps**	**License**
Graphtage	https://github.com/trailofbits/graphtage	Python 3 (cross-platform)	GNU Lesser General Public License
Description	Graphtage is a command-line utility and underlying library for semantically comparing and merging tree-like structures, such as JSON, XML, HTML, YAML, plist, and CSS files. It allows a human to quickly determine the minute differences between large files.		

Tool Name	**URL**	**Platform or Deps**	**License**
Fickling	https://github.com/trailofbits/fickling	Python 3 (cross-platform)	GNU Lesser General Public License
Description	Fickling is a decompiler, static analyzer, and bytecode rewriter for Python pickle object serializations, often used to encode Machine Learning model files. Fickling can take pickled data streams and decompile them into human-readable Python code that, when executed, will deserialize to the original serialized object. It can also detect malicious pickle files, as well as inject code into existing pickles.		

Kudu Dynamics / LevelUp Research

Tool Name	**URL**	**Platform or Deps**	**License**
SPARCLUR	https://github.com/levelupresearch/sparclur	Python 3 (cross-platform), Arlington DOM, Checker, Ghostscript, MuPDF, PDFCPU, PDFium, PDFMiner, Poppler, QPDF, XPDF,	GNU Lesser General Public License
Description	SPARCLUR (Sparclur) is a collection of various wrappers for extant PDF parsers and/or renderers along with accompanying tools for comparing and analyzing the outputs from these parsers.		

Northrop Grumman

Tool Name	**URL**	**Platform or Deps**	**License**
SafeDocs API	N/A	Cross Platform with Java, Java (12), Docker (24), MongoDB (3.12.11), Spring Boot (2.7.3), JeroMQ (0.5.2), Springfox Boot (3.0.0), Junit Jupiter (5.9.2), Apache Common Cli (1.5.0), Jacoco (0.8.8)	Owned by SafeDocs
Description	SafeDocs API manages the SafeDocs tools, database, and exposes http endpoint that serve as CLI endpoints, endpoints for the user interface, and any other client that wishes to interact with the SafeDocs tools.		

Tool Name	**URL**	**Platform or Deps**	**License**
SafeDocs UI	N/A	Cross Platform with any modern browser (Firefox, Chrome) emotion/react: 11.9.0, emotion/styled: 11.8.1, mui/icons-material: 5.6.0, mui/material: 5.6.0, mui/x-data-grid: 5.12.0, testing-library/jest-dom: 5.16.3, testing-library/react: 12.1.5, axios: 0.26.1, ramda: 0.28.0, react: 18.0.0, react-dom: 18.0.0, react-hook-form: 7.29.0, react-redux: 7.2.8, react-scripts: 5.0.0, web-vitals: 2.1.4 babel/core: 7.20.12, babel/plugin-syntax-jsx: 7.18.6, babel/preset-env: 7.20.2, babel/preset-react: 7.18.6, testing-library/user-event: 14.4.3, babel-jest: 29.4.1, babel-preset-jest: 29.4.2, cross-env: 7.0.3, jest: 29.4.3, react-docgen: 5.4.3, react-docgen-markdown-renderer: 2.1.3, react-test-renderer: 18.2.0, ts-jest: 29.0.5	Owned by SafeDocs
Description	SafeDocs UI serves the User Interface to the tool's operator; allowing the operator to select one of the many tools developed for SafeDocs, providing an interface to upload input files and manages the output files by providing filtering and searching capabilities.		

American University

Tool Name	**URL**	**Platform or Deps**	**License**
Dowker tools for statistical inference	https://github.com/kb1dds/dowker_statistics	R >= 3.4, tidyverse	Apache 2.0
Description	Tools for inferring cluster identity using multi-variate Boolean random variables.		

Tool Name	**URL**	**Platform or Deps**	**License**
Python code for topological analysis of (mathematical) relations.	https://github.com/kpewing/relations	Python >= 3.0, numpy	Apache 2.0
Description	Topological distance between two mathematical relations, represented by binary matrices showing which objects (columns) exhibit which features (rows), can be defined in terms of the number of changes required to transform one matrix into the other and vice versa, somewhat like the Hamming distance for strings.		

Tool Name	**URL**	**Platform or Deps**	**License**
Statistical hypothesis tests for exploring relational data	https://github.com/kb1dds/relation_stats	Python >= 3.0, numpy	Apache 2.0
Description	This repository contains two Jupyter notebooks and two standalone Python command line tools to examine the structure of a binary relation		

BAE Systems

Tool Name	**URL**	**Platform or Deps**	**License**
VALARIN file classifier framework for Curator	N/A	Linux, Docker engine>=23.01, Docker-compose >= 2.15, Python3 virtual environment, numpy==1.25.1, pandas==2.0.3, PyYAML==6.0.1, requests==2.31.0	Proprietary
Description	The format-independent VALARIN framework integrates diverse parsers to detect potentially malicious files as part of Curator, the National Geospatial Intelligence Agency's file intake pipeline.		