Implementing a vertically hardened DNP3 control stack for power applications

Sergey Bratus
Dartmouth College
6211 Sudikoff Laboratory
Hanover, NH 03755
sergey@cs.dartmouth.edu

Adam J. Crain
Automatak, LLC
400 W North St
Raleigh, NC 27603
adamcrain@automatak.com

Sven M. Hallberg
Hamburg University
of Technology
Am Schwarzenberg-Campus 3
Hamburg, Germany
pesco@khjk.org

Daniel P. Hirsch
Upstanding Hackers, Inc.
9300 Jaclaire Lane
Anchorage, AK 99502
tq@upstandinghackers.com

Meredith L. Patterson
Upstanding Hackers, Inc.
9300 Jaclaire Lane
Anchorage, AK 99502
mlp@upstandinghackers.com

Maxwell Koo
Narf Industries
PO Box 96503 #37005
Washington, DC 20090
maxk@narfindustries.com

Sean W. Smith
Dartmouth College
6211 Sudikoff Laboratory
Hanover, NH 03755
sws@cs.dartmouth.edu

ABSTRACT

We present an assurance methodology for producing significantly more secure implementations of SCADA/ICS protocols, and describe our case study of applying it to DNP3, in the form of a filtering proxy that deeply and exhaustively validates DNP3 messages. Unlike the vast majority of deployed proprietary DNP3 implementations, our code demonstrates resilience to state-of-the-art black-box as well as white-box fuzz-testing tools.

1. INTRODUCTION

DNP3 [7] is a complex ICS protocol commonly used throughout the US power grid. Recent security investigation of DNP3 production implementations [5] revealed that most of these implementations were vulnerable to attacks via malformed payloads, due to a wide variety of input validation bugs. Out of dozens of commercial implementations, only a few were free of critical defects. These vulnerabilities had dire implications, as they affected the master controllers rather than merely devices in substations [12], and could thus have a devastating effect on the stability of the power grid as a whole. Altogether, over 30 CVEs have been reported for DNP3 in 2013–2014.

Whence the vulnerabilities? We undertook a study of these vulnerabilities and concluded that a new breed of DNP3 parsers was necessary to mitigate these defects. We also came to the conclusion that the root causes of these defects were, in part, due to the syntactic complexity of the protocol, and of the misreadings of the protocol's syntactic complexities by implementors. Even though the DNP3 standards strive for simplicity and clarity, their unwitting choices of certain syntactic constructs undermined these intentions and resulted in preventable bugs. Notably, implementations without defects were those that implemented conservative subsets of the DNP3 specification.

When approached as a formal language to be parsed, DNP3 payloads turned out to have syntax slightly but crucially different from what the structural diagrams in the specification implied, with essential validity requirements hidden in text accompanying these diagrams or occurring elsewhere in the standards documents. It became clear to us that many validity requirements for the DNP3 payloads that could have been expressed in syntactic terms, and checked up front—before the crafted bug-triggering malformed payloads could penetrate deeper into the control systems' logic—were not clearly identified by the standards as such. As a result, easily checkable malformations pass the initial "sanity checks" and cause crashes and potentially exploitable memory corruptions.

Whacking vulnerabilities with better validation. To address these issues, we developed and employed a new software assurance methodology for designing protocol parsers. Our method begins with a careful, *critical reading* of the standard that elicits the full set of requirements that can be validated at the syntax-checking stage—before control functionality is exposed to malicious inputs. We proceed with a *grammatical implementation* of the parser so that it can enforce these requirements correctly and exhaustively before any other processing.

We call the derived enhanced syntax of protocol messages that captures the maximum of possible information about the message objects' *boundaries*, *embedding*, as well as *legitimate occurrence* and *co-occurrence* the **core syntax** of the protocol.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

This view is separate from what these objects *mean* and what actions they should prompt; it is purely structural. Conversely, it aims to capture *all* structural validity requirements for protocol payloads, so that they could be checked early and systematically.

Once this enhanced syntactic view of the protocol is captured, the parser code can be written to clearly correspond to the grammar describing this core syntax. This has the effect of making it clear to both the programmer and the code auditor exactly *what expected input properties have been checked so far, and which ones are being checked.* This approach very effectively mitigates a notorious source of security bugs: the mismatch between the programmer assumptions about the enforced preconditions on the input data vs the actual properties of such data.

In the course of this work, we identified within the current DNP3 specification a number of the syntactic pitfalls that put its implementations at risk. We suggest how to mitigate these pitfalls.

Bringing best-of-breed parsing to ICS embedded systems. We implement our DNP3 parser using our Hammer parser construction toolkit. To the best of our knowledge, Hammer is the first toolkit that brings the security and maintainability benefits of *combinatory parsing* to production C/C++ build environments. Thus, our DNP3 parser is the first to realize this promising approach in embedded and ICS development.

Isolating the improved parser. Besides hardening our input-validating DNP3 parser, we also mitigated any unforeseen errors in it by isolating the parser code from the rest of the control application, and the rest of the application from the raw input buffers, now made exclusively accessible to the parser, and only to the parser. Thus the application's control logic was precluded from accidentally accessing maliciously crafted and unvalidated data in a raw input buffer via, say, a memory pointer. This isolation was achieved via our *ELFbac* intra-process memory access policy, leveraging separation of code and data units of the control application at the Application Binary Interface level. [3, 4]

The structure of the paper is as follows. We first describe our core syntax extraction methodology, based on our *LangSec* approach. We then summarize the vertical hardening of the control system's runtime, based on an ARM Linux industrial computer. Finally, we evaluate the robustness of our system w.r.t. state-of-the-art fuzzers using both generic fuzzing (with American Fuzzy Lop) and DNP3-specific fuzzing (using the custom Aegis tool). We conclude with recommendations for production deployment of our methodologies.

Summary of contributions. To the best of our knowledge, our work is a first in assurance of ICS protocols in several methodological respects. In particular, it is the first work in the ICS domain to:

- base the implementation of an ICS protocol on the parser combinator approach and use it to generate functional unit-testing of the implementation;
- analyze the syntax of an ICS protocol from the Chomsky hierarchy point of view and apply this analysis to preventing security flaws in the implementation;
- apply the security design pattern that enables the policy of isolating the parser from the control application logic;
- use an intra-process memory protection policy in a control application, providing additional guarantees of enforcing intended behavior beyond SELinux, with our novel ELFbac policy mechanism.

As a consequence of the above, fuzz-tesing of our DNP3 input-handling code demonstrated superior resilitency compared to such code in the majority of commercial DNP3 products.

2. APPLYING LANGSEC TO DNP3: A DEEPER UNDERSTANDING OF THE PROTOCOL

LangSec is a view on software security that focuses on identifying and handling inputs safely. [14] It gives rise to a development methodology that posits that the design of input-handling software must start with analyzing the grammar of the protocol, gathering all syntactic requirements under this grammar, and writing the parsing/validating code to resemble the grammar as closely as possible. The latter is best achieved by employing the *parser combinator* approach,[1] which has been implemented for a variety of production languages, including C++, Java, Python, etc. For our DNP3 case study, we used the Hammer parser construction kit,[2] designed specifically for the needs of C/C++ programmers, but also offering bindings for other languages.

Simply put, LangSec posits that **all structural requirements to protocol payloads are syntax**, and must be validated as such, systematically and with the right recognizers for the grammar, before any processing occurs.

The LangSec viewpoint further holds that more syntactically complex protocols are liable to lead to buggy and exploitable implementations. When the nature of validity checks for protocol messages is unclear, the likelihood increases of an implementation failing to check what subsequent code assumes. Therefore, LangSec recommends syntactically simpler formats that stay within the regular or context-free classes of languages.[3] Syntactic elements of a protocol that introduce context-sensitivity must be avoided at the design stage, handled with utmost care if they must be implemented—and, if possible, filtered out from the input language when already deployed. A LangSec analysis of a protocol tends to point out these elements as pitfalls, and a number of famous vulnerabilities suggest that this analysis is correct. [11]

We applied this methodology to DNP3, coming up with a grammatical, combinator-based parser implementation from the protocol specification. In doing so, we also covered the specification's prose requirements wherever these touched on mutual co-occurrence or relationships between elements. From our reading of the specification and during implementation, we discovered that many structural requirements are not described as syntax and do not appear in the diagrams. Instead they are hidden away in the text of the standard, sometimes listed between rules concerning the behavior of an implementation or the effects of a message.

We view all *structural* validity requirements as *syntactic* validity ones. Thus any requirements concerning the interpre-

[1] See [8] for a practical description, and [16, 10] for functional programming foundations of parser combinators.
[2] Designed and implemented by the 4th and the 5th authors.
[3] I.e. those that can be fully parsed by a finite state machine or a pushdown automaton, respectively.

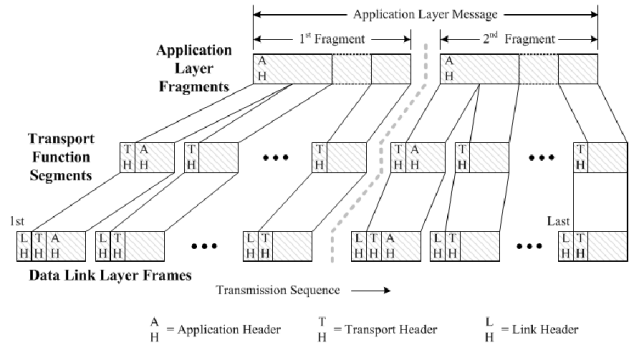


Figure 1: DNP3 protocol layers (Fig. 0-2 in IEEE 1815-2012)

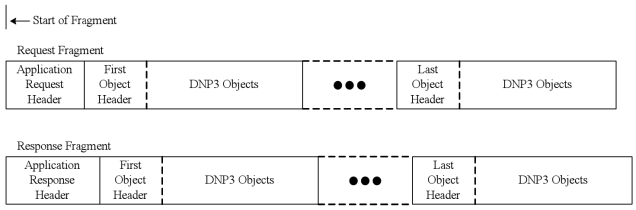


Figure 2: DNP3 fragment structure (Fig. 4-4 in IEEE 1815-2012)

tation of payload object boundaries and embedding, as well as whether an object may or may not occur in a particular context should be extracted, and formalized in the recognition front-end. We stress that this not only minimizes exposure of the rest of the code to unwarranted assumptions, but also has strong potential to warn of ambiguities and pitfalls in the specification text. Indeed, during this process of grammatical formalization, we found a number of problem spots that corresponded to many vulnerabilities found in DNP3 parsers by previous fuzzing efforts.

Several new combinators were added to or implemented on top of the Hammer combinator library, some to support DNP3 specifically, others of general utility.

2.1 Extracting core DNP3 syntax

At the outset, diagrams of DNP3 packets such as Figures 1 and 2 made their syntax look clear enough, hardly more complex than that for the TCP/IP family. DNP3 consists of a simple three-layer protocol stack, each layer adding headers to a higher-level payload.[4]

An application-level message may stretch over multiple packets, called *fragments*. A fragment's high-level structure, as shown in Fig. 2, consists of the application header, followed by a variable number of data "objects", which are grouped by type, each group preceded by a common object header. This structure could be described by the following regular expression:

$$\text{AppHeader } (\text{ObjectHeader Object}^*)^*$$

As we will see, of course, this view is far from complete.

[4]While our implementation covers all three layers, and the link and transport layers have interesting properties in their own right, we focus on the rich application layer structure for our exposition.

The application header consists of control flags, a fragment sequence number, and most importantly the *function code*, a number that specifies the kind of action to take upon receipt of the message. A special function `RESPONSE` is reserved for returning data from an outstation, e.g. from reading a sensor. Here we meet the first complication. In response messages only, the application header includes an additional field called *internal indications* (IIn), which the outstation uses to report faults and various other conditions. The allowed combinations of control flags also differ between requests and responses. Thus the two types of application headers can be described with the following grammar rules:

$$\begin{aligned} \text{ReqHdr} &\rightarrow \text{SeqNo ReqFlags ReqFun} \\ \text{RspHdr} &\rightarrow \text{SeqNo } \mathbf{\text{RspFlags RspFun IIn}} \end{aligned}$$

This is close to our implementation, though the whole truth includes further special cases for the function codes `CONFIRM` and `UNSOLICITED_RESPONSE`.

In addition to header fields, the function code determines allowed and expected types of data objects to follow. Some functions expect no objects, others have complex requirements, and one of the most common functions (`READ`) takes object *headers* only. It is worth noting that official pseudocode published to clarify the parsing process fails to properly account for the `READ` case.[5] In our approach we individually develop grammatical descriptions of the object structure for each function code, allowing us to handle wildly differing rules at function granularity. Duplication is avoided by the use of custom combinators.

Unfortunately, much of the standard is written at odds to our view that the function code determines message structure. Instead, requirements on the context in which objects can occur are attached to the descriptions of the object types. For example, for object group 50, time and date, section A.23.1.2.3 implies that they can appear in read requests and responses; it goes on to state that they may also appear in write requests. As per the discussion above, this information is part of our *core syntax*, and we encode it in the grammar rules for the `READ`, `WRITE`, and `RESPONSE` functions. The supplementary Table 3 of AN2013-004b summarizing the mapping of object types to function codes helped to clarify their expected grammatic relationship; we essentially inverted this table, uncovering some omissions in the process.[6]

Relationships between function codes and their objects can be more complex. The `SELECT` function, for instance, initiates output actions, which follow three basic forms. Whereas analog signals are supported through a straight-forward set of objects (group 42), binary outputs such as relays are controlled either through a complex structure called CROB or a variant consisting of "pattern control" objects (PCB/PCM). From the descriptions of the `SELECT` function code and the aforementioned object types, we arrive at the following gram-

[5]AN2013-004b states in section 2.4.2 that "no object data [is] to parse" only for range specifier code (RSC) 6, "all". For the common case of requesting values from certain indexes, RSC 0–5 and 7–9, it calls for the fragment to be discarded when there is no object data.

[6]E.g. group 120, variant 1, authentication challenge, for `RESPONSE`; group 120, variants 3 and 9, aggressive-mode authentication, for `CONFIRM` (cf. Fig. 7-16 in [7]).

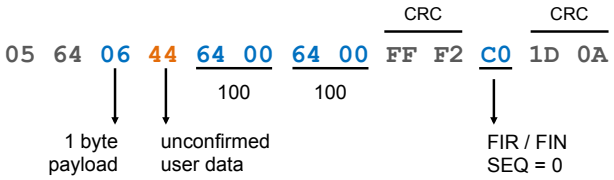


Figure 3: A crafted APDU-less link-layer packet from [5].

mar for a SELECT request's object arguments:

$$\begin{aligned} \text{Select} &\rightarrow \text{Action}^* \\ \text{Action} &\rightarrow \text{AnalogOut} \mid \text{Crob} \mid \text{PcbPcm} \\ \text{PcbPcm} &\rightarrow \text{Pcb Pcm}^+ \end{aligned}$$

Note how one or more PCM objects must follow a PCB.[7] Apart from that, we allow the three types of actions to be mixed freely and also allow empty requests (no actions).

The grammatical approach starts paying off even at a high level. One of the flaws found by [5] was that a packet with correct link- and transport-layer headers but zero-length APDU (application layer message) would cause unhandled exceptions in certain implementations. Such a packet (with correct checksums) is depicted in Fig. 3. The specification requires a transport layer payload to consist of no less than one byte, implying that at least one APDU should be present, but the relevant checks were apparently forgotten by some implementors—a structure-checking vulnerability of omission or lapse of attention.

By contrast, our approach requires explicitly describing the packet's structure at every level. It thus refocuses the programmer's attention on the particulars of the structure, making errors by omission less likely.

The parser code that we then write is little more than the specification of this core syntax—in C/C++ code that is ready for compilation.

2.2 Code that looks like the grammar

The hallmark of the LangSec approach to protocol implementation is that the input-handling code charged with structural validation of incoming payloads literally *looks like the grammar* of the valid inputs. Such code has tremendous advantages for both the programmer and the auditor: it is succinct, and at every line and indeed at every expression, it is obvious what portion of input and which of its properties are being validated.

The above principle guided our implementation. We used the Hammer parser construction kit to both describe the extracted core syntax of DNP3, and, at the same time, to produce our validating parser for this syntax.

We give two examples to illustrate the resulting code style. The first, in Fig. 4, shows the parsing code for the application header, including the cases for confirmations and unsolicited responses. It should be clear from the figure that the code essentially matches the grammar shown earlier. It expresses

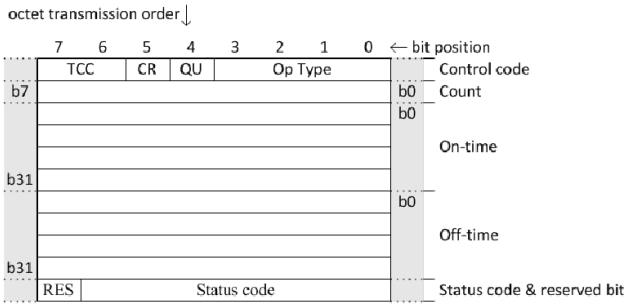
[7]This is a good example where a structural (i.e. syntactic) requirement appears in text that seems otherwise concerned with semantics or behavior. From the description of object group 12, variant 2, pattern control block: "*Executing controls is initiated by sending* . . . a single [PCB] object followed by one or more Pattern Mask objects" (emphasis ours).

```
seqno = h_bits(4, false /* unsigned */);
conac = h_sequence(seqno, conflags, NULL);
reqac = h_sequence(seqno, reqflags, NULL);
unsac = h_sequence(seqno, unsflags, NULL);
rspac = h_sequence(seqno, rspflags, NULL);
iin   = h_sequence(h_repeat_n(bit, 14),
                   reserved(2), NULL);

req_header =
    h_choice(h_sequence(conac, confc, NULL),
             h_sequence(reqac, reqfc, NULL), NULL);

rsp_header =
    h_choice(h_sequence(unsac, unsfc, iin, NULL),
             h_sequence(rspac, rspfc, iin, NULL), NULL);
```

Figure 4: The Hammer-based code for handling the DNP3 application headers.




```
crob = h_sequence(h_bits(4, false),  // op type
                  bit,               // queue flag
                  bit,               // clear flag
                  tcc,
                  h_uint8(),         // count
                  h_uint32(),        // on-time [ms]
                  h_uint32(),        // off-time [ms]
                  status,            // 7 bits
                  reserved(1),
                  NULL);
```

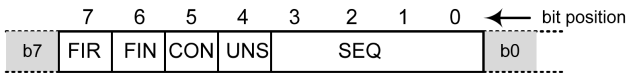
Figure 5: The DNP3 CROB object (IEEE 1815-2012) and the Hammer-based code for handling it.

the requirements on the header structure, at the same time specifying this structure explicitly.

The second example in Fig. 5 is the structure of the CROB objects that directly control binary outputs. This group 12 object consists of a variety of fields describing actions such as opening or closing a relay or producing a strobing signal of a given duration and periodicity. Again, the code shown both parses CROB objects and validates that they have correct structure.

This coding style is used for all fields and syntactic elements from the ground up, starting with individual bit flags. Whereas their semantics is outside the parser's scope, their sequence and any variations are described in the same amount of detail as the more complex objects. For instance, Fig. 6 shows the handling of the bit flags.

If anything, this code is *more* concise than typical C/C++ binary parsing code based on pointer arithmetic and bitmask matching. Yet it is also a lot more readable, and less likely to lead the programmer or the auditor to overlook or misinterpret a syntactic check, leading to Heartbleed-like vulnerabilities.




```
                /* --- uns,con,fin,fir --- */
conflags = h_sequence(bit,zro,one,one, NULL); // CONFIRM
reqflags = h_sequence(zro,zro,one,one, NULL); // fin,fir
unsflags = h_sequence(one,one,ign,ign, NULL); // UNSOL'TD
rspflags = h_sequence(zro,bit,bit,bit, NULL);
```

Figure 6: The handling of flags in the Hammer-based code.

```
pcb    = dnp3_p_g12v2_binoutcmd_pcb_oblock;
pcm    = dnp3_p_g12v3_binoutcmd_pcb_oblock;
anaout = dnp3_p_anaout_oblock

select_pcb    = h_sequence(pcb, h_many1(pcm), NULL);
select_oblock = h_choice(select_pcb, crob, anaout, NULL);
select        = h_many(select_oblock);
```

Figure 7: Defining the SELECT function in Hammer.

2.3 Refining the DNP3 specification

The above approach has the power to warn the programmer of the specification pitfalls and ambiguities that may lead to flaws in structural validation of inputs. It makes all structural validity requirements explicit, whereas a casual reading may leave them masked, leading to vulnerabilities. At the same time, it exposes opportunities and needs for clarification of the standard.

The SELECT function, introduced in Section 2.1, presents an example. Recall the grammar given earlier. It immediately poses the following questions:

1. Are empty SELECT requests considered valid? Should they be passed to the processing code or rejected outright during input validation?

2. Can the same request contain multiple PCB-and-PCM blocks?

3. May a request mix PCBs, CROBs, and analog outputs?

Our methodology results in the code shown in Fig. 7, again mirroring the grammar closely. Alternative answers to the above questions would be easy to implement by making simple modifications.

For another example, we consider the case of the CTO object. Responses may carry objects that include a relative timestamp, e.g. group 2, variation 3, binary input event—with relative time. The specification states that "a preceding common time-of-occurrence (CTO) object, group 51, establishes the basis of the relative time." This seemingly implies that the CTO object is required; however, many current implementations are known to accept relative timestamps without a CTO—using some default as the reference. Moreover, the specification does not state whether multiple CTO objects are permitted within the same message, whether the CTO should *immediately* precede timestamped objects, or whether or where objects without relative timestamps are permitted. This corresponds to a grammar like the following for the objects of a RESPONSE message:

$$\begin{aligned} \text{Response} &\rightarrow \text{RspObject}^* \\ \text{RspObject} &\rightarrow \text{RelativeEvent} \mid \text{Other} \mid \text{CTO} \\ \text{RelativeEvent} &\rightarrow \ldots \text{RelativeTime} \\ \text{Other} &\rightarrow \ldots \textit{(no timestamp)} \end{aligned}$$

The above is what our parser currently implements in order to support all existing deployments. However, guarding against a missing CTO object becomes an obvious requirement after this analysis. Prior to it, overlooking it was easy, and has indeed led to implementation flaws [5]. In addition, the stated ambiguities should be clarified. Different options easily present themselves. For instance, requiring the CTO to be grouped immediately before a block of relative-time objects would look similar to the SELECT case:

$$\begin{aligned} \text{Response} &\rightarrow \text{RspObject}^* \\ \text{RspObject} &\rightarrow \text{Other} \mid \text{withCTO} \\ \text{withCTO} &\rightarrow \text{CTO RelativeEvent}^+ \end{aligned}$$

A maximally permissive grammar enforcing only the requirement that a CTO be present *somewhere* before any relative timestamp can also be realized:

$$\begin{aligned} \text{Response} &\rightarrow \text{Other}^* \text{ withCTO}^* \\ \text{RspObject} &\rightarrow \text{RelativeEvent} \mid \text{Other} \\ \text{withCTO} &\rightarrow \text{CTO RspObject}^* \end{aligned}$$

Thus the LangSec approach not only results in readable parsers that make it obvious which input properties are checked by every statement and every function call, but also warns the implementor of the potential ambiguities in the standard *and* naturally suggests options for clarification to standards authors.

3. SYSTEM ARCHITECTURE SUMMARY

The primary principle of our study was to implement *vertical integration* of security measures: a set of features distributed across the systems components and supporting each other to implement a security policy operating from the kernel boot time throughout the control application's runtime.

In particular, we combined our intra-process ABI-based memory protection technology of *ELFbac*, our input-validation assurance methodology of *LangSec*, with the proven *Grsecurity/PaX* Linux kernel hardening.

Each of these technologies can be applied independently, but they work best in concert. ELFbac [4] enforces programmer intent with respect to data flows between the application's intra-memory code and data units on the ABI level, whereas Grsecurity/PaX's UDEREF feature [15] enforces such intent between the kernel and the userspace. In turn, the LangSec methodology of application design ensures that the security-critical units that handle input validation can be properly separated to take the best advantage of ELFbac policies to mitigate potential exploits targeting these units via crafted inputs.

In order to showcase the composition of these approaches, we built a Linux-based filtering proxy for the DNP3 protocol. This proxy applies *exhaustive inspection* of the DNP3 frames before passing them between an outstation and a master controller; by this we mean full inspection of all syntactic elements of the protocol, such as headers and objects, and all

relationships between them specified by the protocol documentation that can be expressed in the form of a grammar. This methodology brings the absolute majority of the DNP3 frame validity checks forward, into the input-checking unit we call the *recognizer*; this unit is then isolated by an ELFbac policy from the rest of the processing.

The choice of the filtering proxy as the test application for the stack policies was made with the view towards generality: our proxy code is meant to be extensible to other applications that act on DNP3, such as rewriting proxies and adapters to other protocols. Such extensions would replace the processing part but keep the input-validating, parsing part of our code.

Working with a prototype that nevertheless implemented over 70–80% of the underlying complex DNP3 protocol specification in a functional, testable way gave us a complex enough structure to test our security policies.

4. EVALUATION

4.1 Robustness evaluation

We subjected our proxy to fuzz-testing by both the *Aegis* generational DNP3 fuzzer developed by Adam Crain and Chris Sistrunk—with which they demonstrated that most commercial DNP3 implementations were vulnerable to attacks via crafted inputs—and with the *American Fuzzy Lop* fuzzer (AFL) from Michał Zalewski, a state-of-the-art coverage-guided fuzzer.

These two fuzzers utilize two complementary approaches. Aegis [2] uses generational models of DNP3 payloads and was designed for black-box testing of DNP3 implementations. AFL [18], on the other hand, applies genetic algorithms guided by code coverage to a set of seed inputs, and is otherwise protocol-agnostic.

Our baseline for comparison of our implementation with the existing industry ones was the study of Crain & Sistrunk [5], summarized below. We used that study as a proxy for comparing black-box fuzzing outcomes.

It should be noted that these other implementations had the benefit of being deployed in the field and had thus been exposed to at least some level of abnormal inputs even prior to being fuzz-tested with Aegis. It is reasonable to assume that such prior exposure resulted in reporting and fixing of at least some bugs before Aegis had been brought to bear. By contrast, our implementation stood up to fuzzing freshly "out of the box", just after passing its functional unit tests—and nevertheless showed better resiliency.

Additionally, we used AFL to complement the above black-box fuzzing with white-box fuzzing. White-box fuzzing could not provide us with a comparison point with proprietary industry implementations, since it would require access to their sources. However, surviving state-of-the-art white box AFL fuzzing, which has found major vulnerabilities in dozens of critical and well-used software projects,[8] is itself a significant achievement.

Generational fuzzing with Aegis. Despite vigorous testing, our application did not suffer from any bugs beyond the classic resource-exhaustion attack that we subsequently fixed. Although we hoped for exactly this result due to our parser construction methodology, **we note that few commercial implementations approached it in Crain's and Sistrunk's testing [5]—and those that did, unlike ours, implemented small and restrictive subsets of the DNP3 protocol.**

[8]See the "bug-o-rama trophy case" section of [18] for the major bug discoveries attributable to AFL.

```
Frame defect codes:

A:  Specified data not present in object header
B:  Integer boundary condition or near-boundary condition
C:  Bogus APDU function code
D:  Bogus object or qualifier
E:  Bogus qualifier code
F:  Unexpected addressing (broadcast, self, etc)
G:  Unexpected object & qualifier combo
H:  Unexpected object data
I:  Undersized APDU (0, 1, 2 w/o IIN)
J:  Unexpected object and function
K:  Invalid link-layer control block
L:  Link-layer size/function/payload mismatch

Vulnerability codes:

INF: Infinite or long looping
MEM: Looping causing excessive memory allocation
RAC: Read access
BOV: Buffer overrun
UEX: Unhandled exception
UNK: Unknown

Summary of Selected Flaws:

    1  - O { INF } A B
    2  - O { RAC }     C D E F
    3  - O { UEX }           F
    4  - M { INF } A B         G
    5  - M { INF } A B
    6  - M { BOV }             G H   J
    7  - M { UEX }                 I
    8  - M { INF } A B
    9  - O { INF }             G
    10 - M { INF } A B
    11 - M { UNK } A B
    12 - O { UNK }             G     J
    13 - M { MEM } A B           H
    14 - M { RAC } A B
    15 - O { INF } A B               J
    16 - M { INF } A B
    17 - O { UNK }                     K
    18 - M { RAC } A B
    19 - O { UNK }                       L
```

Figure 8: Selected flaws detected in proprietary DNP3 implementations by black-box Aegis fuzzing.

For comparison, we present a selection of flaws found by Aegis black-box fuzzing of over 20 distinct DNP3 products, of which only 2 weren't found to have detectable faults. The vast majority of the products with faults had multiple distinct issues that were amalgamated into a single ICS-CERT security advisory. Overall, 31 public advisories were produced.[9]

Figure 8 shows a summary of selected flaws, classified by the kinds of crafted structural defects in the payload that triggered the flaw, and by the observed effects of triggering the flaw on the target (a Master controller `M` or an outstation `O`). The summaries are presented in encoded form; for example, `M { INF } A B` means that a Master (`M`) is put into an infinite loop (`INF`) by a crafted frame that specifies the presence of certain payload data but lacks it (`A`), and violates a boundary condition (`B`).

Note that all of these flaws have their root cause in failure to structurally validate protocol inputs.

[9]The full list of published advisories is available at https://automatak.com/robus/

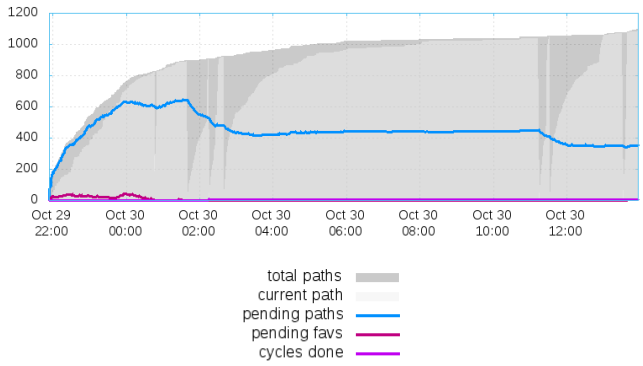


Figure 9: AFL asymptotic path coverage after approx. 16 hours.

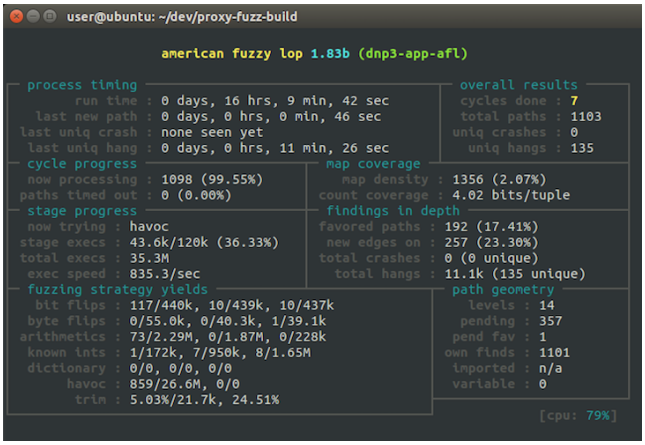


Figure 10: AFL fuzzer screenshot.

Coverage-guided fuzzing with AFL. Fig. 9 shows that we ran AFL to approach the asymptotic limit for discovering new paths given the seed data we provided. No crashes were recorded. Fig. 10 shows a screenshot of the AFL console after approximately 16 hours of fuzzing. We verified that the "hangs" reported in the screenshot are all false positives, due to a low AFL default timeout for heuristically determining a "hang".

Unit test suite. We used a suite of functional unit tests to test our implementation, and assessed the the completeness of the unit test suite via `gcov` coverage. The results for the core parser are shown in Fig. 11.

4.2 Unit-testing & validation methodologies

Whereas fuzz-testing described above provides an empirical evaluation of the system's overall robustness, the individual components must have their respective unit-testing and validation methodologies that can be applied to them in isolation. The following describes how our designs provide for it.

Source directory	Coverage	
src	1107 / 1225	90.4%
src/obj	465 / 465	100.0%

Figure 11: Coverage of the core parser, via `gcov`.

ELFbac policies.

An ELFbac policy is essentially meant to contravene any accesses by code units to data units not allowed by the policy. Since both code and data units are, in fact, ABI units of the executable, the relative positions of either the reference or the referent within their respective units do not matter, so long as these are located within their boundaries.

Consequently, the easiest and most comprehensive way to test an ELFbac policy's efficacy is to insert memory references to disallowed sections into its code units, and assert the resulting memory traps.

Representing the policy's allowed accesses as a bipartite labeled graph between the code sections and the data sections of an executable, the exhaustive test is easily derived as the complement of that graph. Specifically, for each specific edge of this complement graph, we constructed a unit test for the policy, by placing synthetic violating instructions at the top of the particular code section.

Moreover, dereferencing pointers in the wrong context is a known security concern. One additional power of ELFbac is that, even though it does not control the passing of pointers to system calls and program units, it should trap attempts by wrong code units to dereference these pointers if they point to an ELFbac-labeled data section that contravenes the access policy. We tested this behavior by placing contravening dereferences synthesized according to the above graph.

LangSec parsers.

The LangSec methodology for constructing input-validating parsers lends itself to exhaustive unit-testing by design. Namely, under this methodology, parsers for the whole protocol messages are constructed from the parsers for their simpler parts, and so on, down to the simplest elements such as integer and string fields. Consequently, for every protocol unit from the primitive types up, a definitive function that validates these elements and these elements alone exits, and can be tested independently of all others.

In other words, the structural units of the parser correspond to the structural units of the grammar. In turn, these correspond to natural unit tests for the functions implementing their parsing and validation.

Validation and resource control.

Small payloads in DNP3 (and other ICS protocols) can result in construction of large objects, consuming resources on the receiving endpoint. Thus a resilient application must resist resource consumption attacks.

From testing our application both via fuzzing and known-bad payload scenarios, we concluded that building the parser as a modular structure allows flexibility w.r.t. semantic object representation (which may be resource-intensive). Creation of these representations must be explicitly a part of the semantic actions, separate from the recognizer.

This encourages moving all possible checks that logically precede the creation of an object into the recognizer—which is where it really belongs, since no objects should be constructed (or acted upon) until the incoming data that describes them is validated.

Performance overhead.

We functionally tested our proxy in the setting of filtering single DNP3 sessions over TCP. In this context, we found performance overhead of parsing negligible compared to other I/O costs, and well within the bounds for the targeted application: protecting an outstation or a master from a single outstation (or a small number of outstations).

More rigorous performance testing would be required for the scenario of filtering all of a master's traffic, which can reach over 2000 concurrent sessions. In this scenario, sheer throughput may matter, and a different platform may be required to handle it. Similar concerns apply to real-time protocols such as GSSE, which has latency requirements of under 4ms. This work is ongoing.

DNP3 traffic and encryption.

The DNP3-SAv5 proposal introduces selective cryptographic authentication for a subset of DNP3 function codes. In particular, SA specifies a set of codes that must always be authenticated, and a subset that can be optionally authenticated. The rationale for the selective authenticated design was to conserve bandwidth, but we believe it also ushers in a dangerous security anti-pattern. In [6] we discuss the additional attack surface DNP3-SAv5 adds, and the threat models it overlooks.

5. RECOMMENDATIONS FOR COMMERCIAL DEVELOPMENT

The ELFbac model is recommended for the industry developers who seek to leverage both the existing Linux ecosystem and the best-of-breed Linux kernel self-protection of Grsecurity/PaX, while remaining compatible with the standard C/C++ ABI and its build chain. ELFbac is complementary to the mandatory access controls (MAC) such as SELinux, which operate at the granularity level of an entire process, and can be combined with an ELFbac intra-process policy without any additional costs.

More specifically, SELinux policies apply to system calls issued by a process entirely based on the process' SELinux identity label, regardless of the order of issue or of the particular code executing in the process at the time of the system call. For example, an allowed system call can be issued either by the main executable or any of its loaded libraries, at any stage of the process' timeline; it is all the same to SELinux.[10]

In short, SELinux and other MAC schemes concern themselves neither with the order of system calls, nor with the order of memory accesses that happen inside a process. For control processes that naturally contain distinct phases of operation, this level of control is clearly not expressing the programmer's intended operation of the process.

ELFbac is the only policy of its kind that allows the developer to enforce the *intra-process* access control between the structural units of a program at its runtime—such as between libraries loaded into the process and their sensitive data to which these libraries are intended to have exclusive access. Further, ELFbac's intra-process policies isolate sensitive code units from accidentally operating on untrusted, un-validated, potentially maliciously crafted data, and the sensitive data units from being accessed or operated upon by code units not intended to do so.

[10]For this reason, SELinux permission policies are colloquially described as a "bag of permissions", per process.

Thus we recommend the ELFbac protections for any front-end ICS/SCADA systems facing untrusted data. Although effective use of ELFbac requires that the policy expresses some knowledge of programmer's intent w.r.t. the program's units—such as the sequence in which these units are expected to execute and the kinds of data they are supposed to execute on—these intents are typically easily observable at the development time and well-understood by the programmers.

An explicit enforcement mechanism for these intents will both help the programmer catch errors and help communicate these actual intents to Unix runtime, where they are currently almost entirely ignored. Thus ELFbac captures in a few lines per compilation and/or scoping units of a program what modern programming languages strive to achieve via new language semantics: enforceability of existing and clear intent at runtime. However, unlike most programming languages, ELFbac maintains compatibility with the core C/C++ ABI, binary OS utilities, and the build chain tools.

Leveraging Grsecurity/PaX, a state-of-the-art kernel hardening technology. Grsecurity/PaX is the industry's leading kernel hardening technology. Meant for general-purpose systems, the Grsecurity/PaX patch is considerably more complex than it needs to be for control systems that do not need, e.g., to support just-in-time compilation (JIT) at runtime and can place other restrictions on the applications as a matter of policy.

A version of Grsecurity/PaX offering a cohesive subset of its protective features [1] has been ported by the Grsecurity/PaX team for the ARM industrial computer platform used in our project.

We integrated our ELFbac kernel mechanism with it without conflict, thus leveraging the strongest practical kernel self-protection mitigations in industry into our prototype hardened stack.

6. RELATED WORK

Our LangSec methodology builds on many insights and efforts in systems and network security. A comprehensive survey of these is beyond the scope of this paper. We refer the reader to [14] for such a survey, and to [9] for recent research on secure parsing through verification. Wang et al. [17] and Pike [13] explore a different but closely related approach based on embedded Domain Specific Languages (eDSL) for embedded control systems.

Conclusion

We demonstrated that input-checking parsers for complex ICS protocols such as DNP3 can be written robustly and succinctly by starting with a critical reading of the protocol specification and extracting all structural requirements on the protocol payloads into a grammar, then by writing the parsing code to closely follow this grammar. In the process of such reading and implementation, troublesome ambiguities of the specification are naturally exposed and fixed.

Our implementation of a DNP3 validation proxy withstood vigorous fuzz-testing with state-of-the-art tools—which *few commercial implementations survived in prior tests [5]. Moreover, those few implemented small and restrictive subsets of the DNP3 protocol compared to our implementation.*

We demonstrated that ABI-based intra-memory protection policies do not place a considerable burden on ICS programmers, so long as they follow reasonable design practices.

We also demonstrated that our approach composes with the state-of-the-art Linux kernel hardening technology, Grsecurity/PaX.

Code availability

The code of our DNP3 parser construction kit and the exhaustive core syntax validation proxy are available via *http://langsec.org/dnp3/* and on Github as *https://github.com/pesco/dnp3* and *https://github.com/sergeybratus/proxy* respectively, under the BSD license. The code of the ELFbac ARM implementation is available at *https://github.com/sergeybratus/elfbac-arm/*, with the supporting materials at *http://elfbac.org/*.

Acknowledgments

This work was supported in part by the Intel Lab's University Research Office, by the TCIPG project from the Department of Energy (under grant DE-OE0000097), and by the Schweitzer Engineering Laboratories Inc., who made the ARM port of ELFbac for ICS systems possible. Its continued development is supported by the CRED-C project and the DARPA RADICS program.

7. ADDITIONAL AUTHORS

8. REFERENCES

[1] Grsecurity® for ICS. https://grsecurity.net/ics.php.

[2] Automatak. Aegis fuzzer for ICS/SCADA. https://www.automatak.com/aegis/.

[3] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E. Locasto, Jason Reeves, Sean W. Smith, and Anna Shubina. ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection. Dartmouth Technical Report TR2013-272, 2013 June. http://www.cs.dartmouth.edu/reports/TR2013-727.pdf.

[4] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Jason Reeves, Sean W. Smith, Anna Shubina, Maxwell Koo, and Michael E. Locasto. Intra-Process Memory Protection for Applications on ARM and x86: Leveraging the ELF ABI. Black Hat USA, 2016 August. http://elfbac.org/bh16-elfbac-whitepaper.pdf.

[5] Adam Crain and Chris Sistrunk. Project Robus, Master Serial Killer. Digital Bond's SCADA Security Scientific Symposium (S4x14), January 2014.

[6] J. Adam Crain and Sergey Bratus. Bolt-On Security Extensions for Industrial Control System Protocols:A Case Study of DNP3 SAv5. *IEEE Security & Privacy*, 13(3):74–79, May–June 2015.

[7] DNP Technical Committee. 1815-2012 - IEEE Standard for Electric Power Systems Communications-Distributed Network Protocol (DNP3). http://ieeexplore.ieee.org/document/6327578/, October 2012. Revision of IEEE Std 1815-2010.

[8] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide, Second Edition*. Springer, 2008. pp. 564–566.

[9] Adam Koprowski and Henri Binsztok. TRX: A Formally Verified Parser Interpreter. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 345–365, 2010.

[10] Daan Leijen. Parsec, a fast combinator parser. http://research.microsoft.com/en-us/um/people/daan/download/parsec/parsec.pdf, October 2001.

[11] Falcon D. Momot, Sergey Bratus, Sven Hallberg, and Meredith L. Patterson. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *IEEE Cybersecurity Development Conference (IEEE SecDev)*, November 2016.

[12] Dale Peterson. Why Crain / Sistrunk Vulns Are A Big Deal. https://www.digitalbond.com/blog/2013/10/16/why-crain-sistrunk-vulns-are-a-big-deal/, October 2013. Digital Bond.

[13] Lee Pike. Hints for High-Assurance Cyber-Physical System Design. In *Proceedings of IEEE Cybersecurity Development (SecDev)*. IEEE, November 2016. Preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/sedev16.html.

[14] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Security Applications of Formal Language Theory. *IEEE Systems Journal*, 7(3):489–500, September 2013.

[15] Brad Spangler. The UDEREF feature, mailing list excerpts. https://grsecurity.net/~spender/uderef.txt, 2007. Part of Grsecurity/PaX suite, https://grsecurity.net/.

[16] S.D. Swierstra. Combinator Parsers: From Toys to Tools. In *Electronic Notes in Theoretical Computer Science*, volume 41, pages 38–59, August 2001.

[17] Yan Wang and Verónica Gaspes. An Embedded Language for Programming Protocol Stacks in Embedded Systems. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*, 2011.

[18] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.