Editors: Patrick McDaniel, mcdaniel@cse.psu.edu | Sean W. Smith, sws@cs.dartmouth.edu

Bolt-On Security Extensions for Industrial Control System Protocols:

A Case Study of DNP3 SAv5

J. Adam Crain | Automatak
Sergey Bratus | Dartmouth College

Industrial control system (ICS) protocols—key to public utility operations—have developed alongside the Internet but are largely isolated from it, carried by dedicated serial lines between closed networks with trusted software. However, as leased lines are replaced with transmission control protocol (TCP) or wireless connections to serve the needs of "smarter" energy systems and as ICS traffic comingles with other kinds of packets, legacy ICS protocol design becomes a problem. Protocols previously designed for isolated networks must receive "bolt-on" security extensions, compatible with the bulk of legacy implementations already deployed; implementations never meant to be exposed to maliciously malformed input must be hardened to reject it gracefully. Attempting to realize visions of smart utilities with the current ICSs' attack surfaces will dramatically increase risks of their catastrophic failure due to hostile actions.

DNP3 (IEEE Standard 1815-2012) is widely used in the US power grid and is a typical representative of the supervisory control and data acquisition (SCADA)/ICS protocol family.[1] As with many other such protocols, DNP3's original design didn't include security features such as authentication. The recently standardized secure authentication (SA) extends DNP3 to provide optional and multiuser authentication services, with characteristic tradeoffs between security and bandwidth. These extensions modify the existing DNP3 application layer by creating additional function codes and object types that selectively apply to a subset of protocol features, leaving others unprotected and increasing overall syntactic complexity.

We believe that this manner of extension represents a security *anti-pattern*—a design that will keep producing bugs and weaknesses—and considerably increases the attack surface associated with protocol encoding, parsing, and implementation complexity. Reviews of SA have overlooked this additional attack surface, focusing instead on its cryptographic primitives and message flows. In this article, we discuss this increased attack surface and how to avoid its worst pitfalls.

DNP3 Overview

The DNP3 protocol stack is split into three layers: link, transport, and application (see Figure 1). The protocol is transport agnostic—all three layers are used regardless of whether the underlying network is a serial communications channel or a TCP stream (with its own open systems interconnection model layers below DNP3's link).

The link layer is concerned primarily with framing, point-to-multipoint addressing, and error detection in a manner similar to Ethernet datagrams, but it also includes simple stateless functionality such as heartbeat messages.

The transport layer reassembles multiple link layer frames into larger application messages. This reassembly is based on a single-byte transport header with first, final, and sequence parameters that allow for only in-order reassembly. Unexpected transport segments are simply dropped, and the reassembly buffer is emptied. The maximum default size of a reassembled

Copublished by the IEEE Computer and Reliability Societies

application layer message is 2,048 octets, although this size is adjustable if both ends agree on the value of using out-of-band configuration.

The application layer handles messages called *application data service units* (ASDUs) that derive their semantics from a combination of function codes and objects. Messages can consist of zero or more object headers that follow the main application layer header (see Figure 2). Object headers describe the type and quantity of objects that follow. The beginning of the next header is discoverable only by parsing the previous one.

The rules for determining object payload lengths are complex and varied. This complexity gives implementations of the DNP3 application layer a large attack surface due to potential programmer errors. For example, programmers might fail to check a payload's multiple object lengths for consistency, interpret a payload's contents differently than intended, or assume the presence of objects that are actually absent from a maliciously crafted payload. As usual in software exploitation scenarios, acting on incorrect assumptions while allocating or copying maliciously crafted payload data results in memory corruptions, which attackers can leverage to crash or control ICS processes.

Most types of valid messages require at least one object header. Notable exceptions are `confirm`, `cold restart`, `warm restart`, `delay measurement`, and `record current time`, which are never paired with any objects. The specification exhaustively defines which objects can be paired with which function codes.[1]

The vast majority of object headers can be processed independently—that is, they aren't context sensitive with regard to other object headers in the ASDU. Nonsecure DNP3 has only one notable exception to this rule: `common time of occurrence` fields and the measurements with which they can be paired. This combination of headers requires a common reference time header to precede one or more relative times and acts as a crude way of compressing what's normally 48-bit time stamps on measurement values.

In any protocol, there's an inherent design tradeoff between structural flexibility and attack surface, which doesn't favor cryptography. Indeed, underlying complexity or ambiguity of encoding gave rise to a variety of attacks such as Serge Vaudenay's and Daniel Bleichenbacher's as well as the more recent BEAST, CRIME, Lucky13, POODLE, BERserk, and others, which all worked around the enduring strength of the cryptographic primitives.

With its high level of flexibility, the DNP3 application layer is a poor candidate for encoding cryptographic functions. Despite the constraints placed on function and object combinations, the number of valid combinations of objects for many DNP3 function codes is practically infinite. The ability to associate multiple objects to a single function makes the DNP3 application layer powerful in terms of flexibility and bandwidth but also particularly vulnerable when it comes to parsing and processing attacker-supplied input. By contrast, SCADA protocols of similar functionality, such as IEC 60870-5-104, have more rigid application layer structures in which the function code completely defines the type of data that follows, reducing the combinatorial complexity of valid inputs (and thus the complexity of the code that must validate them). Not surprisingly, DNP3's complexity is reflected in its distribution of vulnerabilities.

Fuzzing Vulnerable Implementations

Vulnerabilities in DNP3 implementations that arise due to the protocol's complexity aren't theoretical. For nearly a decade, such vulnerabilities in DNP3 and other SCADA protocol implementations have been found by fuzzing[2,3]; however, little information has been made publicly available on DNP3 vulnerability specifics. A 2010 US government–funded report specifically mentioned the dire need to improve input parsing routines in DNP3 implementations without citing specific failure modes.[4]

The most comprehensive study of DNP3 vulnerabilities was conducted by Crain (coauthor of this article) and Chris Sistrunk from 2013 to 2014 and resulted in numerous disclosures coordinated with vendors and asset owners (www.automatak.com/robus); a small representative fraction of the raw vulnerability data was released publicly.[5] We recap the results of this study here, as they pertain to DNP3 security extensions.

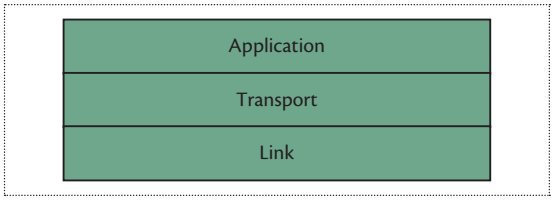

Figure 1. Abstract DNP3 communication stack. The link layer has direct access to the communication channel.

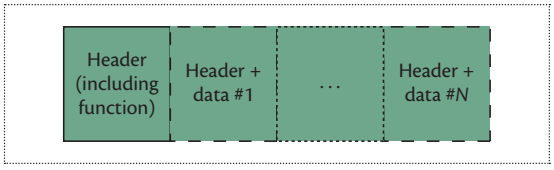

Figure 2. DNP3 application layer messages consist of a main header and zero or more object headers and associated data.

Examples of Vulnerabilities

Crain and Sistrunk tested the effects that crafted malformed frames could have on DNP3 implementations in master controllers and outstation (remote) equipment. Nearly all vendor products were

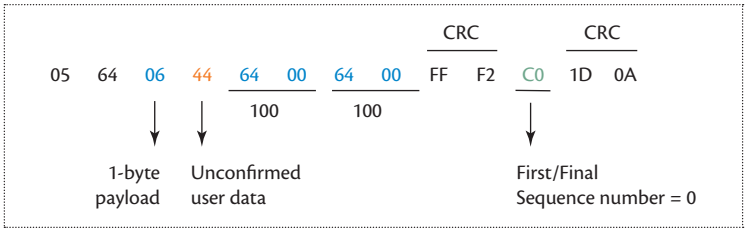

Figure 3. A DNP3 frame with source address 100 and destination address 100. It contains no application layer payload and caused a fault in a real system owing to poor input validation. CRC is cyclic redundancy check.

found to be vulnerable to single-frame attacks for certain frame types; these types were chosen to exercise the protocol's syntactic complexity and to trigger programmer errors that would likely result from this complexity.

A single crafted frame received by a vulnerable implementation could crash the receiving process or drive it into an infinite loop, rendering the entire protocol stack inoperable. Moreover, for many vendors, broadcast frames could trigger such effects, which doesn't require any attacker knowledge about the link endpoint configurations.

For example, ASDUs that are too short to contain a valid object header could be delivered in a frame with a correct lower-layer cyclic redundancy check (CRC) value to cause an unhandled exception in the receiving code. An infinite loop could be exploited in another implementation by setting an object count to the maximum possible value of 65,535 but failing to provide these bytes. A response with two control objects unexpected in such a frame would cause a buffer overrun and crash—an example of a payload that's syntactically valid according to the specification but meaningless. This creates room for ambiguity of payload interpretation. In other cases, malformations in simpler lower layers caused crashes, for example, a link frame encapsulating a single-byte malformed transport protocol data unit and no application protocol data unit (APDU).

A single frame triggering an unhandled exception can be as simple as a payload that contains no APDU under the valid link and transport layer checksums (see Figure 3).

Distribution of Vulnerabilities

The generational fuzzer used in this study was designed to stress each layer of the protocol individually to expose weaknesses in each layer's implementation. The tool was iteratively improved using code coverage analysis obtained from an open source implementation of DNP3.

More than 80 percent of discovered vulnerabilities were found in the application layer. This isn't surprising given how DNP3's complexity is distributed. The DNP3 specification devotes hundreds of pages to describing the application layer, its state machine, and the numerous object encodings, whereas the link layer is covered in only 21 pages and the transport layer reassembly gets a mere seven pages. We find similar ratios by counting the source lines of code associated with each layer in an open source implementation of the protocol. Simply put, when it comes to robustness and security, less is more.

Of the application layer vulnerabilities, a disproportionate number were associated with the unsolicited response functions. A crude way of explaining this is to analyze the specification to see how many object types can be paired with certain function codes. Performing this analysis on the Parsing Guideline Tables in IEEE Standard 1815-2012 reveals that the most overloaded functions in terms of the number of possible object types are `read`, `response`, and `unsolicited response`.[1] The near absence of vulnerabilities in the `read` function code is best explained by the fact that client ASDUs don't carry data payloads but merely describe what data is being requested, resulting in a simpler syntax. `Responses` and `unsolicited responses` can be associated with the majority of the object headers and types in the specification, giving these functions the highest attack surface.

Underrepresentation of the Application Layer

Despite the majority of the failures discovered in the application layer, there's reason to believe that this layer is underrepresented in the results as compared to the link and transport layers. The open source package used to verify the fuzzer is a conservative implementation that doesn't include even more complex protocol feature subsets such as file transfer, datasets, and device attributes. The fuzzer was developed to verify this open implementation and therefore doesn't model these optional features. Many of these features use more complicated encodings that include variable length fields, many of which can be specified in multiple ways and can be internally inconsistent and potentially confusing to a parser. It's almost certain that significant latent vulnerabilities exist in these complex but untested areas of the various protocol implementations.

Optional Authentication

The SA specification lists a set of function codes that must always be authenticated as well as a smaller subset of function codes that can be optionally authenticated. This decision was made to conserve communication bandwidth.[1] However, selective authentication of

application layer messages is a counterproductive and dangerous design pattern, especially in SCADA. Optional authentication conveys a false sense of security to users, fails to address the vulnerability threats posed by parsing and processing payloads, and substantially increases the protocol's overall complexity by requiring security mechanisms to be protocol aware.

Unauthenticated Closed Loop

The spoofing of measurement data has been a component of several major attacks against ICSs, including Stuxnet, allowing attackers to cause more undetected damage or losses to a process over time than with a sudden catastrophic event. In this context, not providing mandatory authentication of measurement data from the field is an important oversight. Man-in-the-middle attackers on an SA link with only mandatory authentication enabled can allow authenticated control information to pass but subtly alter measurement data in such a way that gradually degrades the process or damages equipment.

Lack of Stateless Functionality

Because almost no stateless functionality can be found in the protocol, configuring an SA system to not authenticate any particular function code is inadvisable. The DNP3 application layer has only a handful of completely stateless function codes. The `delay measurement` function code, for instance, doesn't alter any server-side state in the outstation when processed. However, because of the event-oriented nature of DNP3, a combination of `read` and `confirm` functions allows attackers with access to the network to flush all queued event messages from an outstation if these functions aren't authenticated.

DNP3 SAv5 requires the authentication of 21 out of 34 total function codes, whereas the remaining function codes can be optionally authenticated based on the configuration. Mandatory function codes—listed in IEEE Standard 1815-2012[1]—are primarily those that can alter the outstation's state and the process's output state. A notable exception is the `assign class` function code, which can be used to silence an outstation's reporting mechanism by assigning all event data in the outstation to class 0. This would have an effect similar to `disable unsolicited` but could be even more harmful because it would likely persist across device reboots and remove event data from responses to normal event polls.

Responses Present the Most Risk for Exploitation

As we discussed, the most complex `response` and `unsolicited response` function codes present the highest attack surface and therefore the most risk of exploitation. Furthermore, remotely compromising a physically well-protected master from an isolated and less-protected field asset was until recently an underdiscussed attack vector.[6,7]

The attack model under which the specification was designed doesn't seem to include implementation defects as a viable threat. Selectively authenticating subsets of the protocol by function alone—and not for complexity—is a major oversight and should be regarded as a secure protocol anti-pattern.

Conversely, requiring authentication prior to parsing these complex areas of the specification would turn a preauthentication exploit into one requiring compromised credentials.

Protocol Complexity

DNP3 is a complex protocol, mostly due to the way it implements the transfer of event data using server-side state. A lot of bookkeeping and additional messages such as `confirms` are required to keep things synchronized. SA adds even more complexity to the same set of application layer state machines in a manner that's difficult to untangle. This presents real challenges for implementers who must now support both the secure and nonsecure versions of the protocol.

This complexity also extends into parsing and ambiguous encodings. Cryptographic protocols should always defer as much parsing and processing as possible until after the sender's identity has been established to both derive the most benefit from cryptographic integrity protections and avoid the so-called "cryptographic doom principle."[8] Unfortunately, SA's design contradicts this principle.

Challenge–Response versus Aggressive Mode

After an initial session key exchange, normal DNP3 traffic can be authenticated using one of two modes: challenge–response and *aggressive*, which is a form of one-pass authentication using sequence numbers for replay protection.

The challenge–response mode introduces two additional messages into the normal traffic flow and can substantially impact latency and throughput for a serial link. This two-pass authentication mode is more resistant to replay attacks because each message is authenticated using a unique nonce for each challenged message. In this mode, it's fairly easy for the challenging party to treat everything after the function code as opaque "payload data" that isn't parsed until the remote side authenticates. Figure 4 shows challenge–response mode's traffic flow.

Aggressive mode adds a user object with a sequence number as the first object header in the ASDU and a hash message authentication code (HMAC) value as the last object header. The purpose of this mode is to reduce bandwidth and latency by authenticating messages in a request–response exchange. Figure 5 shows the request's structure.

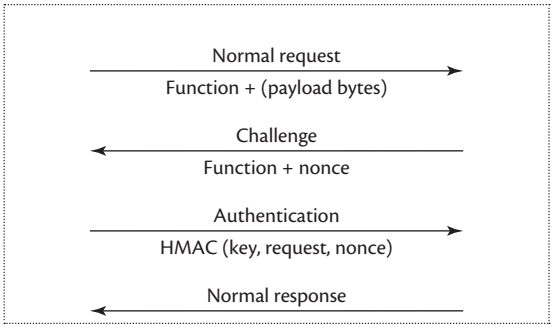

Figure 4. Challenge–response message flow. Parsing of the message payload can be deferred until after authentication. HMAC is hash message authentication code.

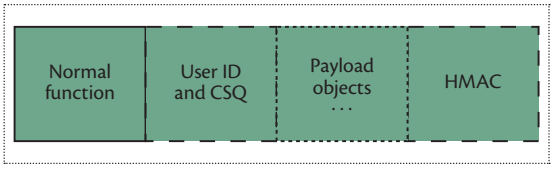

Figure 5. In aggressive mode, application data service units sandwich the payload to be processed inside an ad hoc envelope consisting of user and sequence information and a trailing message digest. The challenge sequence number (CSQ) protects messages from replay attacks.

Aggressive Mode Ambiguity

The first issue with aggressive mode request encoding is the ambiguity of the request. Normally, DNP3 message payloads can be processed solely based on the function code. In aggressive mode, the first object header must be inspected to determine whether the ASDU is a normal request or an aggressive mode request. The lack of a proper envelope for the payload data requires implementers to perform special-case parsing in multiple places to safely handle aggressive mode requests.

The most dangerous issue with aggressive mode encoding is that many implementers will naively parse the entire payload data to reach the HMAC trailer. Recall that DNP3 object headers can't normally be skipped over without at least some level of light parsing. Numerous vulnerabilities were identified in the parsing of these object headers, particularly integer overflow issues related to handling object headers that use start and stop indices. At first blush, it appears necessary to interpret the inner payload data to be able to determine the trailing HMAC's position.

Fortunately, in this case, there's a nonintuitive and undocumented workaround. The HMAC object and its header are of a known size and can be speculatively parsed off the end of the ASDU. Future versions of the specification should make explicit recommendations to implementers to use this methodology for reading the aggressive mode HMAC. We note that the signing schemes for Linux's loadable kernel modules have finally converged on a similar design in which a fixed-size signature is simply appended to the end of the module object file after a string of unsuccessful designs that attempted to use more complex formats and metadata.

Conflicting Encodings of Length

Many variable-length objects related to security functionality have inconsistent encodings between objects as well as encodings with multiple ways of representing the length of certain fields in a single object. Having two sources of truth for lengths of certain payload elements has been a common source of implementation defects in various protocols, most recently OpenSSL's Heartbleed and the GNU TLS Hello bug, as well as classic preauthentication bugs such as OpenSSH's challenge–response vulnerability.

In DNP3, all variable-length objects are preceded by a UINT16 length that defines the entire object's length. Fixed-length fields come first in the object, and variable-length fields come last. All but the last variable-length field is preceded by its own UINT16 length field. The last field's length is implicitly established as the remainder of the envelope length. Figure 6 shows this pattern.

In this encoding, message authentication code (MAC) value length is unambiguous in the sense that there's only one way to determine its value. If the total size of the object is N and the length of all fields preceding the MAC value is P, then the length of the MAC value is $N - P$. However, this encoding scheme isn't applied consistently. Some objects have a preceding length field for the final variable length field, as Figure 7 shows.

Thus, there are two ways to determine the `master challenge data` field's length in an `update key change request`. In a valid encoding of this object, the entire object's length must agree with the final field's explicit length value. To complicate the issue, the specification informs implementers that they can use either method to establish the final field's length,[1] which can lead to implementations that disagree on the cryptographic data's contents. If the protocol can't be redesigned to remove such encoding ambiguities, the parsing recommendation should be to always check that these two methods produce the same length value.

DNP3 SA contains a number of anti-patterns that will likely serve as a significant source of bugs. Vendors and standards bodies adding security to SCADA/ICS protocols should strongly favor a layered approach to security in which legacy protocol issues can be decoupled from SCADA object models and semantics. ■

Acknowledgments

This specification review was performed as part of the process of implementing it in a preexisting open source project. The DHS S&T HOST program award partially funded this work.

References

1. *IEEE Std. 1815-2012*, IEEE Stan-

dard for Electric Power Systems Communications-Distributed Network Protocol (DNP3), IEEE, 2012; https://standards.ieee.org/findstds/standard/1815-2012.html.
2. G. Devarjaran, "Unraveling SCADA Protocols: Using Sulley Fuzzer," DEFCON 15, 2007; www.dc414.org/download/confs/defcon15/Speakers/Devarajan/Presentation/dc-15-devarajan.pdf.
3. D.G. Peterson, "Iccpsic Assessment Tool Set Released," Digital Bond, 2007; www.digitalbond.com/blog/2007/08/28/iccpsic-assessment-tool-set-released.
4. *NSTB Assessments Summary Report: Common Industrial Control System Cyber Security Weaknesses*, tech. report INL/EXT-10-18381, Idaho Nat'l Laboratory, May 2010; http://fas.org/sgp/eprint/nstb.pdf.
5. D. Peterson, "S4x14 Video: Crain/Sistrunk—Project Robus, Master Serial Killer," Digital Bond, 23 Jan. 2014; www.digitalbond.com/blog/2014/01/23/s4x14-video-crainsistrunk-project-robus-master-serial-killer.
6. D. Peterson, "Why Crain/Sistrunk Vulns Are a Big Deal," Digital Bond, 2013; www.digitalbond.com/blog/2013/10/16/why-crain-sistrunk-vulns-are-a-big-deal.
7. E. Byers, "DNP3 Vulnerabilities Part 1 of 2—NERC's Electronic Security Perimeter Is Swiss Cheese," Tofino Security, 7 Nov. 2013; www.tofinosecurity.com/blog/dnp3-vulnerabilities-part-1-2-nerc%E2%80%99s-electronic-security-perimeter-swiss-cheese.
8. M. Marlinspike, "The Cryptographic Doom Principle," Thought Crime blog, 13 Dec. 2011; www.thoughtcrime.org/blog/the-cryptographic-doom-principle.

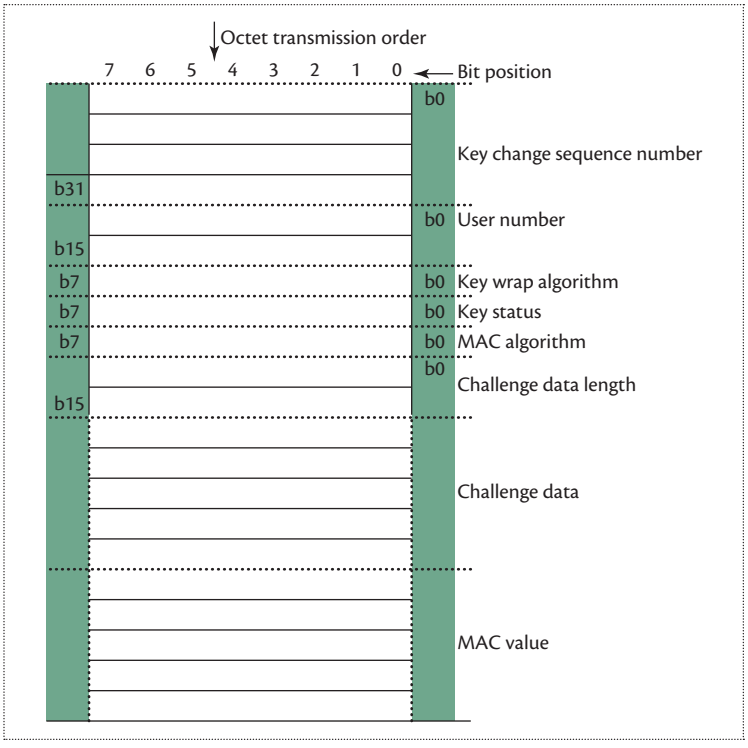

Figure 6. A session key status object with two variable-length fields, challenge data, and message authentication code (MAC) value. The MAC value's length is the remainder of the length field framing the entire object.[1]

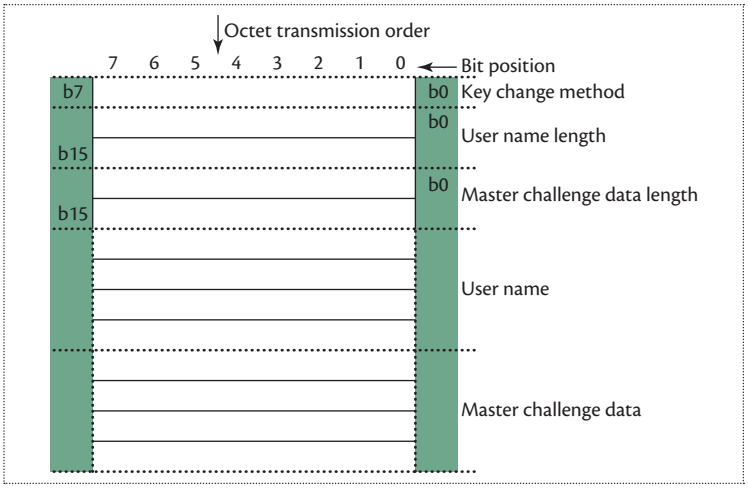

Figure 7. `Update key change request` with two variable-length fields, `user name` and `master challenge data`. The length of the challenge data is explicitly encoded in the length field and implicitly encoded as the remainder of the length field framing the entire object.

J. Adam Crain is a software engineer, security researcher, and open source advocate. He's also a partner at Automatak, which aims to improve the penetration of robust open source software in the utility space. Contact him at jadamcrain@automatak.com.

Sergey Bratus is a research associate professor in the Computer Science Department at Dartmouth College. His research interests include Unix security and wireless networking. Contact him at sergey@cs.dartmouth.edu.