California Energy Systems for the 21st Century (CES-21) Program

Secure SCADA Protocol for the 21st Century (SSP-21)

Adam Crain, Automatak
Note on Public Disclosure

The CES-21 Cybersecurity R&D effort is focused on the protection of critical infrastructure, therefore a secure process for reporting and a secure process for deliverables will need to be maintained. Detailed tactics, techniques, and procedures developed for use fall under DHS guidelines and will be marked and handled as “Protected Critical Infrastructure Information (PCII)” and not open to the public.
The objective of the CES-21 Program is to address challenges of cyber security and grid integration of the 21st century energy system for California through a Collaborative Research and Development Agreement (CRADA). The CES-21 Program utilizes a team of technical experts from Lawrence Livermore National Laboratory (LLNL) and three large Investor-Owned Utilities (IOUs) within the State of California.
How did we determine need for SSP-21?

- **SCADA Protocol Security**
  - **DNP3 (IEEE 1815-2012) Secure Authentication**
    - Can only be used to secure DNP3
    - Complex, too many options, difficult to implement
      - Potential implementation vulnerabilities
      - Interoperability challenges
  - **SSCP (IEEE 1711-2010) Secure SCADA Communications Protocol**
    - AGA-12 Secure SCADA Protection Protocol (SSPP)
    - Data integrity with message authentication and optional encryption
    - STATUS: Withdrawn Standard
  - **Proprietary OEM Solutions**
SSP-21 Requirements

- Simplicity of implementation
- Strongly vetted cryptography
  - Borrow secure compositions from modern, well-studied protocols
- Extensible only to the extent necessary
  - Protocol versions & cipher-suite specifications
- Session message “time of validity”
  - Additional protection for control systems not present in TLS
- Minimal impact of handshake, i.e. preferably 1-RTT
SSP-21 Requirements continued....

- **Basis of trust – Must be public key cryptography!**
  - Modern elliptic-curve Diffie Hellman & signature scheme (25519)
- **Low overhead and processing compared to TLS /RSA / x509**
  - Lower CPU and bandwidth for embedded systems.
- **Optional Encryption (later)**
  - AEAD modes only to reduce chance of misimplementation
- **Support bump-in-the-wire (BitW) retrofits of outstations**
- **Supports serial and IP (can also be used in place of TLS)**
Secure SCADA Protocol for the 21st Century (SSP-21)

Initial protocol specification complete with pre-shared public keys.

Will be made available on Github when legal review is complete.

www.github.com/ssp21/ssp21-spec
SSP21 defines 2 layers

Crypto layer

[ function ] [ function specific payload ]

Link Layer

[ start ] [ destination ] [ source ] [ length ] [ crc-h ] [ payload … ] [ crc-b ]

- Completely stateless
- Provides framing, addressing, and good error detection
- A good default for applications that need such services
Message types and encoding rules

- Message definitions use a lightweight grammar
- A set of simple encoding rules define how to (de) serialize
- There is only one valid encoding of any message
- All messages can be parsed/formatted with only stack allocation
- None of the cruft of ASN.1
  - No embedded length fields (not self-describing)
  - No choice / polymorphism
  - No optional fields
  - Only fixed width integers (uint8_t, uint32_t, uint64_t)
  - No strings types!

```c
enum Function {
    REQUEST_HANDSHAKE_BEGIN    :  0,
    REPLY_HANDSHAKE_BEGIN      :  1,
    SESSION_DATA               :  2
}
```
A full message definition:

Request Handshake Begin

message RequestHandshakeBegin {
  function : enum::Function::REQUEST_HANDSHAKE_BEGIN
  version : U16
  nonce_mode : enum::NonceMode
  handshake_dh : enum::DHMode
  handshake_hash : enum::HandshakeHash
  handshake_kdf : enum::HandshakeKDF
  session_mode : enum::SessionMode
  certificate_mode : enum::CertificateMode
  ephemeral_public_key : Seq8<U8>
  certificates : Seq8<Seq16<U8>>(max = 6)
}

Crypto suite specification
Abstract cryptographic functions

- **Hash**
  - \(\text{HASH(input)} \rightarrow \text{tag}\)

- **Message Authentication Code (MAC)**
  - \(\text{MAC (key, message)} \rightarrow \text{tag}\)

- **Key Derivation Function (KDF)**
  - \(\text{KDF(salt, input_key_material)} \rightarrow (\text{key1, key2})\)

- **Diffie Hellman function (DH)**
  - \(\text{DH(local_priv_key, remote_public_key,)} \rightarrow \text{shared_secret}\)
Concrete cryptographic functions

- **Hash**
  - SHA-256
  - Blake2 ([RFC 7693](https://tools.ietf.org/html/rfc7693))

- **MAC**
  - HMAC SHA-256
  - Keyed Blake2 ([RFC 7693](https://tools.ietf.org/html/rfc7693))

- **Key derivation function (KDF)**
  - **HKDF-SHA256** - HMAC based extract and expand ([RFC 5869](https://tools.ietf.org/html/rfc5869))
  - Blake2 based key derivation

- **Diffie Hellman function (DH)**
  - **Curve 25519** – Fast and secure DH and DSA
  - Curve448 – Wider security margin than 25519
Key Agreement Handshake

Initiator (master) — responder (outstation)

Request Handshake Begin

Reply Handshake Begin

Session Data (s1, n = 0, possibly empty payload)

Session Data (s2, n = 0, possibly empty payload)
Quick Review of Diffie Hellman ...

\[ K_s = dh(kp_1^{\text{priv}}, kp_2^{\text{pub}}) \quad K_s = dh(kp_2^{\text{priv}}, kp_1^{\text{pub}}) \]
Key Agreement

\[ s^i \quad e^i \]

\[ h = \text{hash}(\text{msg1}) \]

\[ h = \text{hash}(h \mid \text{msg2}) \]

\[ e^i.p \quad s^i.p \]

\[ e^r.p \quad s^r.p \]

\[ h = \text{hash} (\text{msg1}) \]

\[ h = \text{hash} (h \mid \text{msg2}) \]

\[ (s_1, s_2) = \text{kdf}(h, \text{dh}(e^r.k, e^i.p) \mid \text{dh}(s^r.k, e^i.p) \mid \text{dh}(e^r.k, s^i.p)) \]

\[ (s_1, s_2) = \text{kdf}(h, \text{dh}(e^i.k, e^r.p) \mid \text{dh}(s^i.k, e^r.p) \mid \text{dh}(e^i.k, s^r.p)) \]
Now we have session keys!

- First message session message in each direction (n = 0) authenticates the sending party to the receiving party.

- Keys are usable for a maximum number of messages
  - Default TBD
  - maximum is the size of the nonce $2^{16} - 1$.

- …. or a maximum duration of time since the start of the session.
  - Default TBD
  - maximum is the width of a uint32_t field as milliseconds (~49.7 days)

- How do we transmit secure session data using these keys?
struct AuthMetadata {
    nonce : U16
    valid_until_ms : U32
}

message SessionData {
    function : enum::Function::SESSION_DATA
    metadata : struct::AuthMetadata
    user_data : SEQ16[U8]
    auth_tag : SEQ8[U8]
}
SSP-21 Reference Implementation

- **SSP-21 Open Source Project**
  - Currently hosted on private GitHub
  - Continuous integration with Jenkins
  - After OSS release, there will be a mailing list

- **BSD3 License**
  - Chosen to encourage broad commercial adoption

- **Validation & Verification of Initial SSP-21 Specification and Reference Implementation**
  - Lawrence Livermore National Laboratory (LLNL)
Technical Highlights

- Lightweight subset of C++ suitable for embedding
  - No heap allocation after initialization
  - No exceptions

- Parsers/formatters for crypto messages generated from models of a grammar.

- Pluggable crypto backend with default of **libsodium**
```c++
void Handshake::derive_authentication_key(
    const seq32_t& message,
    const PrivateKey& priv_s_dh_key,
    const seq8_t& pub_e_dh_key,
    const seq8_t& pub_s_dh_key,
    std::error_code& ec)
{
    this->mix_ck(message);

    DHOOutput dh1;
    this->algorithms.handshake.dh(this->local_ephemeral_keys.private_key, pub_e_dh_key, dh1, ec);
    if (ec) return;

    DHOOutput dh2;
    this->algorithms.handshake.dh(this->local_ephemeral_keys.private_key, pub_s_dh_key, dh2, ec);
    if (ec) return;

    DHOOutput dh3;
    this->algorithms.handshake.dh(priv_s_dh_key, pub_e_dh_key, dh3, ec);
    if (ec) return;

    this->algorithms.handshake.kdf(
        this->chaining_key.as_seq(),
        { dh1.as_seq(), dh2.as_seq(), dh3.as_seq() },
        this->chaining_key,
        this->authentication_key
    );
}
```

Code that looks like the specification
object RequestHandshakeBegin extends Message {

  override def name: String = "RequestHandshakeBegin"

  def function = CryptoFunction.requestHandshakeBegin

  override def fields: List[Field] = List(
    U16("version"),
    Enum(NonceMode()),
    Enum(DHMode()),
    Enum(HandshakeHash()),
    Enum(HandshakeKDF()),
    Enum(HandshakeMAC()),
    Enum(SessionMode()),
    Enum(CertificateMode()),
    Seq8("ephemeral_public_key"),
    Seq8Seq16("certificates", 6)
  )
}

Message DSL and code generator in Scala
struct RequestHandshakeBegin final : public IMessage
{
    virtual ParseError read(seq32_t input) override;
    virtual FormatResult write(wseq32_t& output) const override;
    virtual void print(IMessagePrinter& printer) const override;

    static const Function function = Function::request_handshake_begin;

    IntegerField<openpal::UInt16> version;
    EnumField<NonceModeSpec> nonce_mode;
    EnumField<DHModeSpec> dh_mode;
    EnumField<HandshakeHashSpec> handshake_hash;
    EnumField<HandshakeKDFSpec> handshake_kdf;
    EnumField<HandshakeMACSpec> handshake_mac;
    EnumField<SessionModeSpec> session_mode;
    EnumField<CertificateModeSpec> certificate_mode;
    SeqField<openpal::UInt8> ephemeral_public_key;
    SeqSeqField<openpal::UInt8, openpal::UInt16, 6> certificates;

};

Generates C++ headers and implementation.
Testing and verification

- Continuous integration
- Unit tests
- Code Coverage
- **Cppcheck** SA

In the future:
- Coverity SA for OSS
- Fuzzing w/ AFL
Industrial Key Infrastructure (IKI) Concept

Trust Center

- Trust Center Public Key stored in SCADA Devices during initial configuration
- SCADA Devices generate Public/Private Key Pair to be signed by CSA

Master Certificate Signing Authority

- Certificate Signing Requests (Fast Expiring)
- Certificate Revocation Lists

Outstation Certificate Signing Authority

- Certificate Signing Requests (Long Expiration)
- NO Certificate Revocation Lists

Determine Authenticity of SCADA Devices

- Enterprise & ICS Databases

SCADA Master(s)

- SCADA IED(s)
- SCADA Proxy(s)
- Legacy SCADA Device(s)

Certificate Signing Requests (Fast Expiring)

Certificate Signing Requests (Long Expiration)

NO Certificate Revocation Lists

Trust Center Public Key stored in SCADA Devices during initial configuration

SCADA Devices generate Public/Private Key Pair to be signed by CSA

HSM

Enterprise & ICS Databases

Outstation Certificate Signing Authority
Lightweight certificates needed

- ASN.1 and its encodings BER/DER are too heavy
  - x.509 certificates are too big (multiple kb) for bandwidth limited environments
  - Much of the metadata in x.509 isn’t relevant to utilities
  - Also a hotbed of security issues
    - Cannot typically be parsed without dynamic allocation
    - Variable-length integer fields
    - Too many ways to represent things like strings

- Have looked at x.509 derivatives like M2M format, but they still inherit too much baggage from x.509
Possibly reuse our message grammar!

message CertificateEnvelope {
  certificate_data : Seq8[U16]
  issuer_id : Seq8[U8](count = 8)
  algorithm : enum::SIGNATURE_ALGORITHM
  signature : Seq8[U8]
}

message CertificateData {
  id : Seq8[U8](count = 8)
  valid_after : U64
  valid_before : U64
  type : enum::CERTIFICATE_TYPE
  key_type : enum::PUBLIC_KEY_TYPE
  public_key : Seq8[U8]
  extensions : Seq8[Seq16[U8]]

  Ed25519 / 448
  - 64-bit ids
  - 256-bit 25519 keys
  - 512-bit 25519 signatures
  - No extensions
  - 129 bytes!
SSP-21 updates coming in 2017

- Phases 1 -> 3
  - Specification w/ pre-shared keys (complete)
  - Reference implementation with pre-shared keys (complete)
  - Lab testing of serial BitW with pre-shared keys (in progress)

- Phases 4 -> 6
  - Extend specification with certificate support (mid-Spring)
  - Extend reference implementation and BitW with certificate support (summer)

- Phases 7+
  - Define CA integration and recommended key management architectures
  - Begin working with a standards body like OASIS
SSP-21 Open Source Project

Companies, Researchers, Developers - Participation Welcome!

Contact: rcorrigan@semprautilities.com

Contact: jadamcrain@automatak.com

Contact: ssp21@googlegroups.com