Protocols
Part II

Brian A. LaMacchia
bal@cs.washington.edu
bal@microsoft.com

This material is provided without warranty of any kind including, without limitation, warranty of non-infringement or suitability for any purpose. This material is not guaranteed to be error free and is intended for instructional use only.

Agenda

- Finish up session-based protocols
  - IPSEC Key Management
- Message-based protocols
  - S/MIME
  - XMLDSIG, XMLENC & WS-Security

IPSEC Key Management

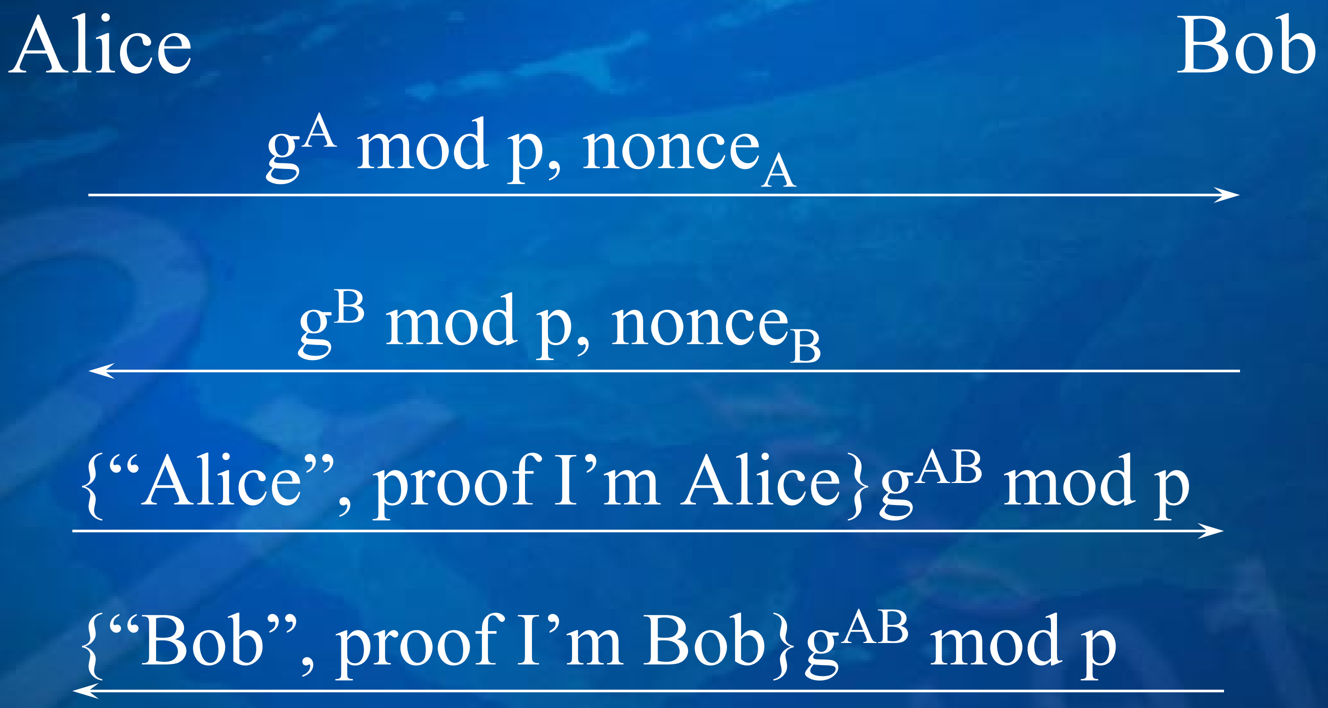
IPSEC Key Management

- IPSEC Key Management is all about establishing and maintaining Security Associations (SAs) between pairs of communicating hosts

Security Associations (SA)

- New concept for IP communication
  - SA not a “connection”, but very similar
  - Establishes trust between computers
- If securing with IPSEC, need SA
  - IKE protocol negotiates security parameters according to policy
  - Manages cryptographic keys and lifetime
  - Enforces trust by mutual authentication

General idea of IKEv2



Alice
Bob
g^A mod p, nonce_A
g^B mod p, nonce_B
{“Alice”, proof I’m Alice}g^{AB} mod p
{“Bob”, proof I’m Bob}g^{AB} mod p

- **It’s just Diffie-Hellman Key Exchange!**

Internet Key Exchange (IKE)

- Resynchronize two ends of an IPsec SA
  - Choose cryptographic keys
  - Reset sequence numbers to zero
  - Authenticate endpoints
- Simple, right?
  - Design evolved into something very complex

IKE Contenders

- Photuris: Signed Diffie-Hellman, stateless cookies, optional hiding endpoint IDs
- SKIP: Diffie-Hellman public keys, so if you know someone's public key g^B, you automatically know a shared secret g^{AB}. Each msg starts with per-msg key S encrypted with g^{AB}
- And the winner was…

ISAKMP

- Internet Security Association and Key Management Protocol
- Gift to the IETF from NSA
- A “framework”, not a protocol. Complex encodings. Flexible yet constraining.
- Two “phases”. Phase 1 expensive, establishes a session key with which to negotiate multiple phase 2 sessions

Internet Key Exchange (IKE)

- Phase I
  - Establish a secure channel (ISAKMP SA)
  - Authenticate computer identity
- Phase II
  - Establishes a secure channel between computers intended for the transmission of data (IPSEC SA)

Internet Key Exchange (IKE)

- IKEv1 authors tried to fit academic papers (SKEME, OAKLEY) into ISAKMP
- Mostly a rewriting of ISAKMP, but not self-contained. Uses ISAKMP
- Since both so badly written, hadn’t gotten thorough review
  - Really 3+ specs (ISAKMP, IKE, DOI)
  - Plus a few more (NAT traversal, etc.)

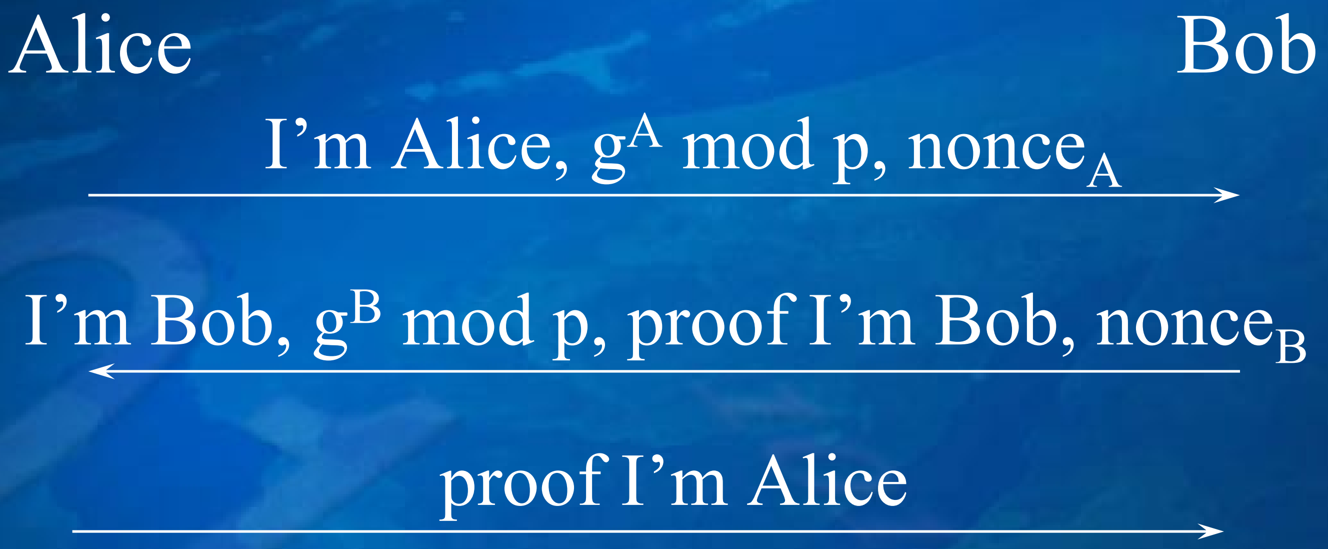
Imagine 150 pages of this!

- While Oakley defines “modes”, ISAKMP defines “phases”. The relationship between the two is very straightforward and IKE presents different exchanges as modes which operate in one of two phases.
  —RFC 2409

IKE

- Two phases, like ISAKMP
- Phase 1 is 8 protocols!
  - Two “modes”: aggressive (3 msgs), and main (6 msgs)
  - Main does more, like hiding endpoint identifiers
- Phase 2 known as “quick mode”
- So 9 protocols (8 for phase 1, + phase 2)

General Idea of Aggressive Mode



Alice

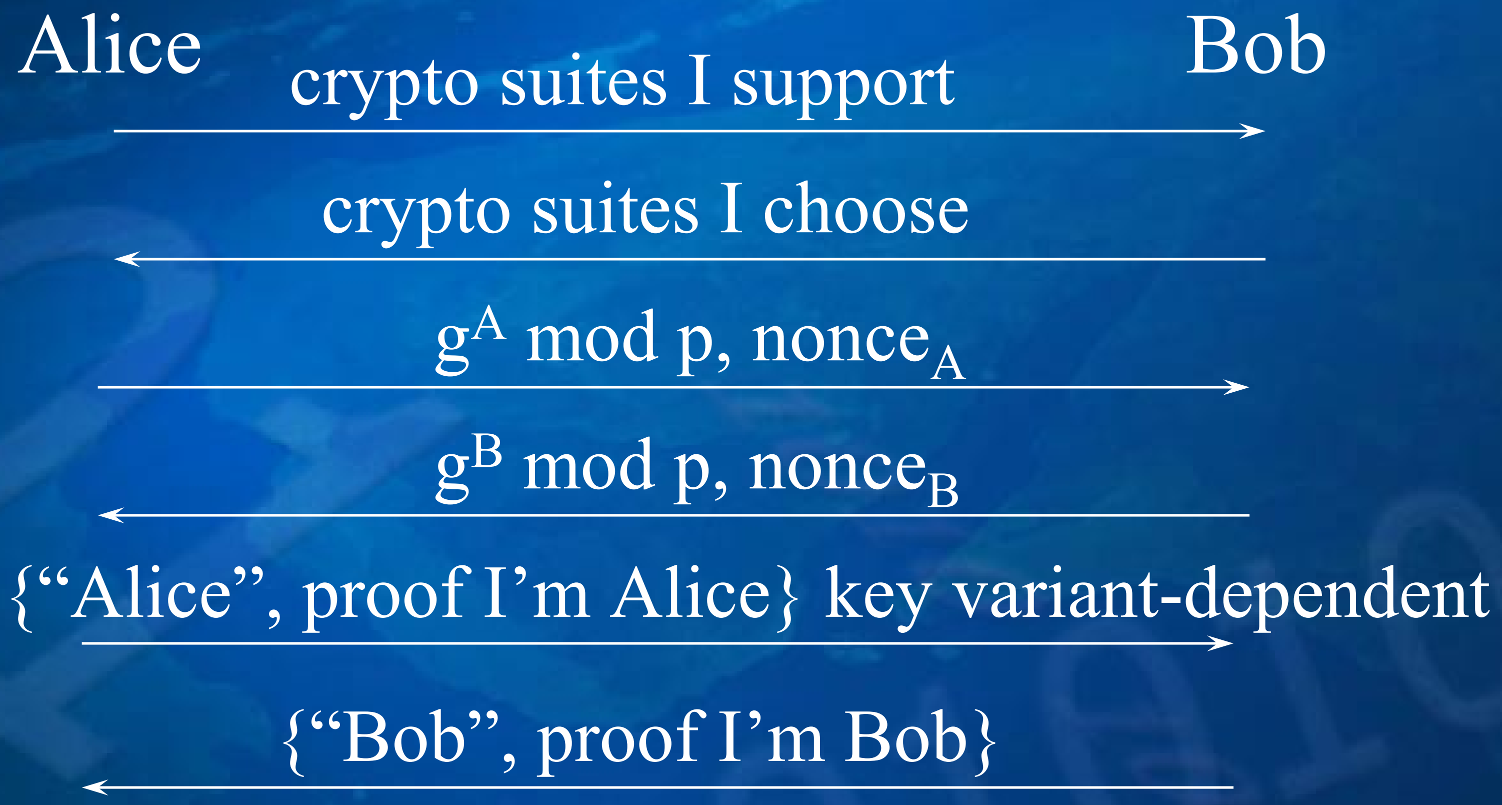
Bob

I’m Alice, g^A mod p, nonce_A

I’m Bob, g^B mod p, proof I’m Bob, nonce_B

proof I’m Alice

General Idea of Main Mode



Alice

Bob

crypto suites I support

crypto suites I choose

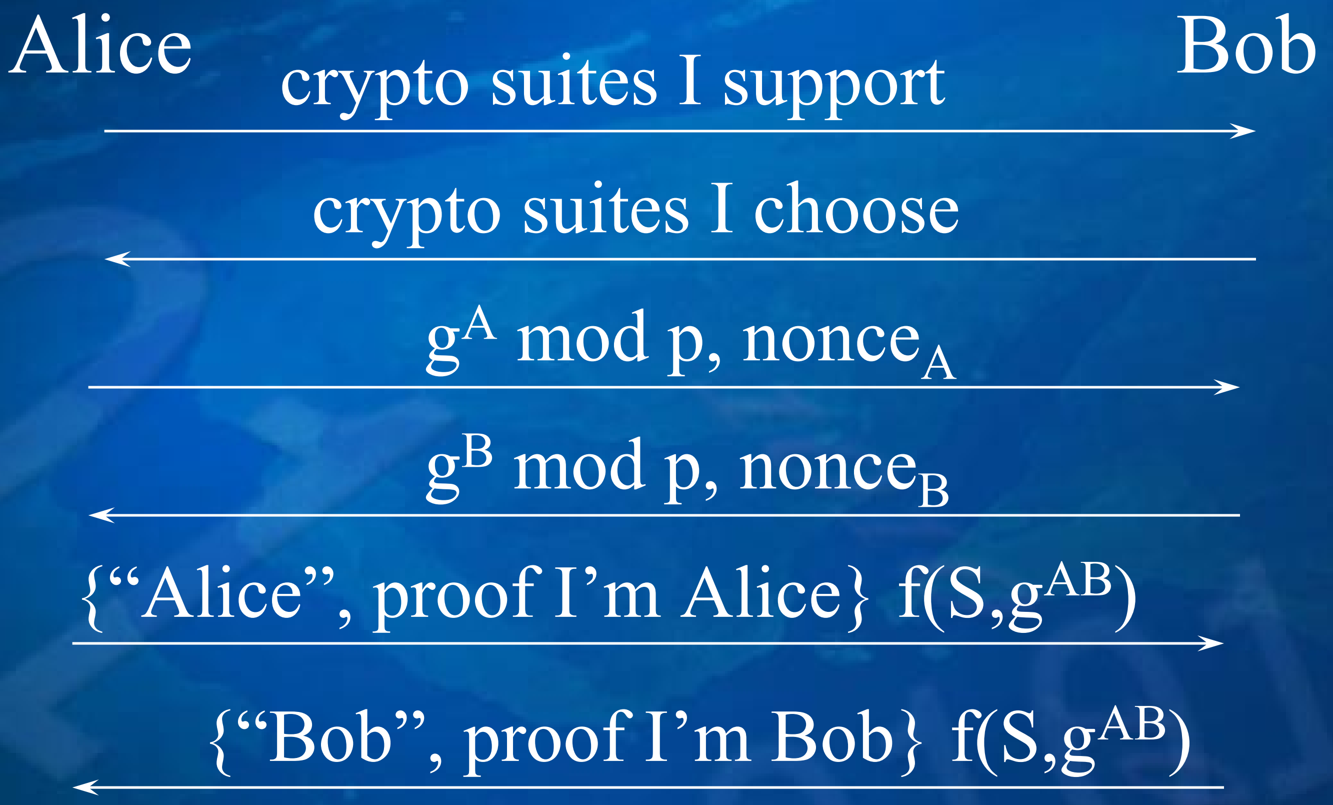
g^A mod p, nonce_A

g^B mod p, nonce_B

{“Alice”, proof I’m Alice} key variant-dependent

{“Bob”, proof I’m Bob}

Main-Mode-Preshared key S



Alice

Bob

crypto suites I support →

← crypto suites I choose

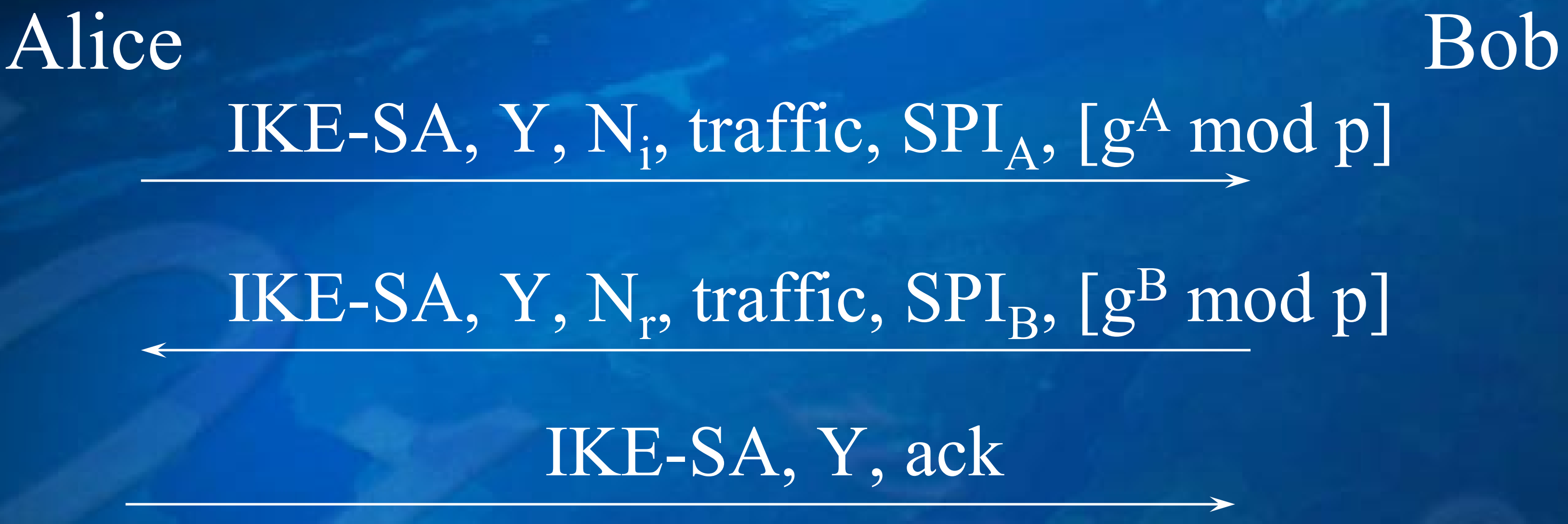
g^A mod p, $nonce_A$ →

← g^B mod p, $nonce_B$

{"Alice", proof I'm Alice} $f(S,g^{AB})$ →

← {"Bob", proof I'm Bob} $f(S,g^{AB})$

General idea of Quick Mode



Alice

Bob

IKE-SA, Y, N_i, traffic, SPI_A, [g^A mod p]

IKE-SA, Y, N_r, traffic, SPI_B, [g^B mod p]

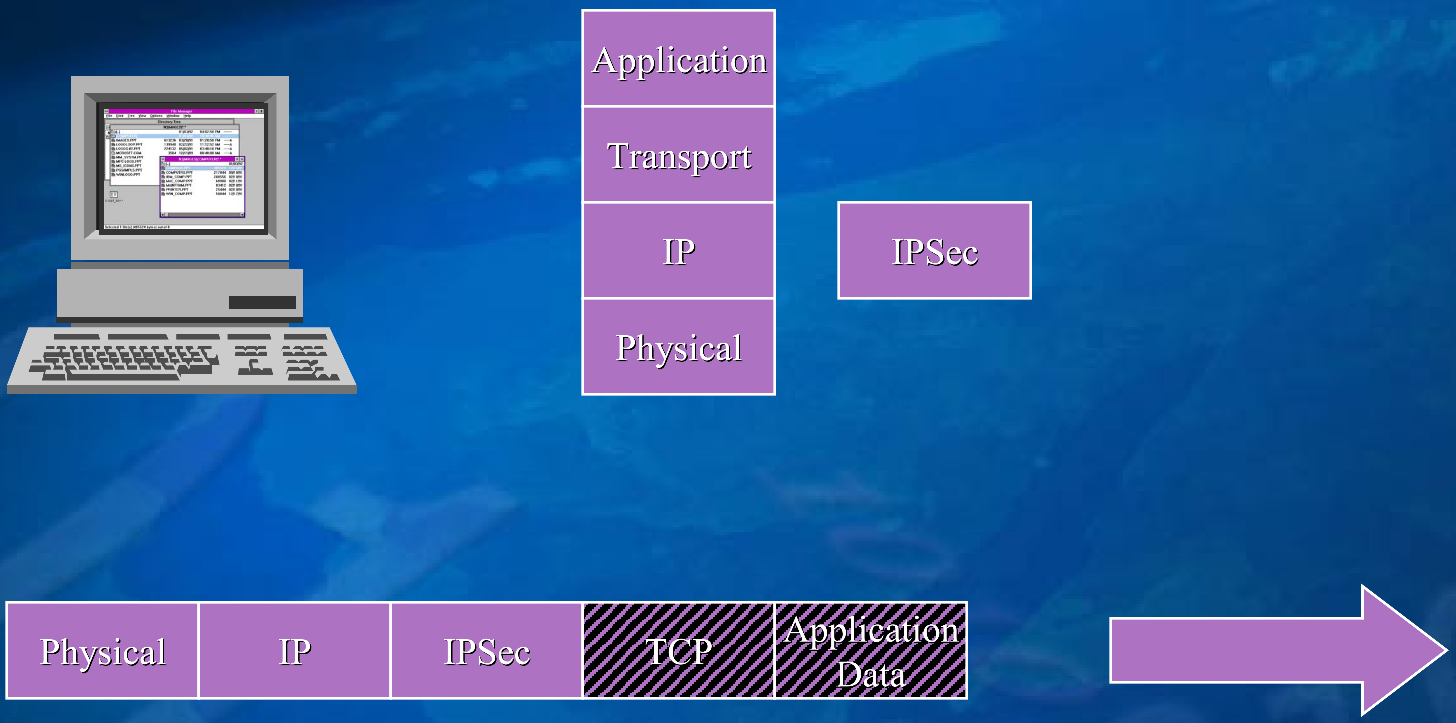
IKE-SA, Y, ack

New key is PRF(current key, g^{AB} | N_i | N_r)

IPSEC Bundling/Wrapping

- Multiple IPSEC transforms may be wrapped successively around a single IP datagram
  - Example: IPSEC transport sent over an IPSEC tunnel

Sending in Transport Mode



Application

Transport

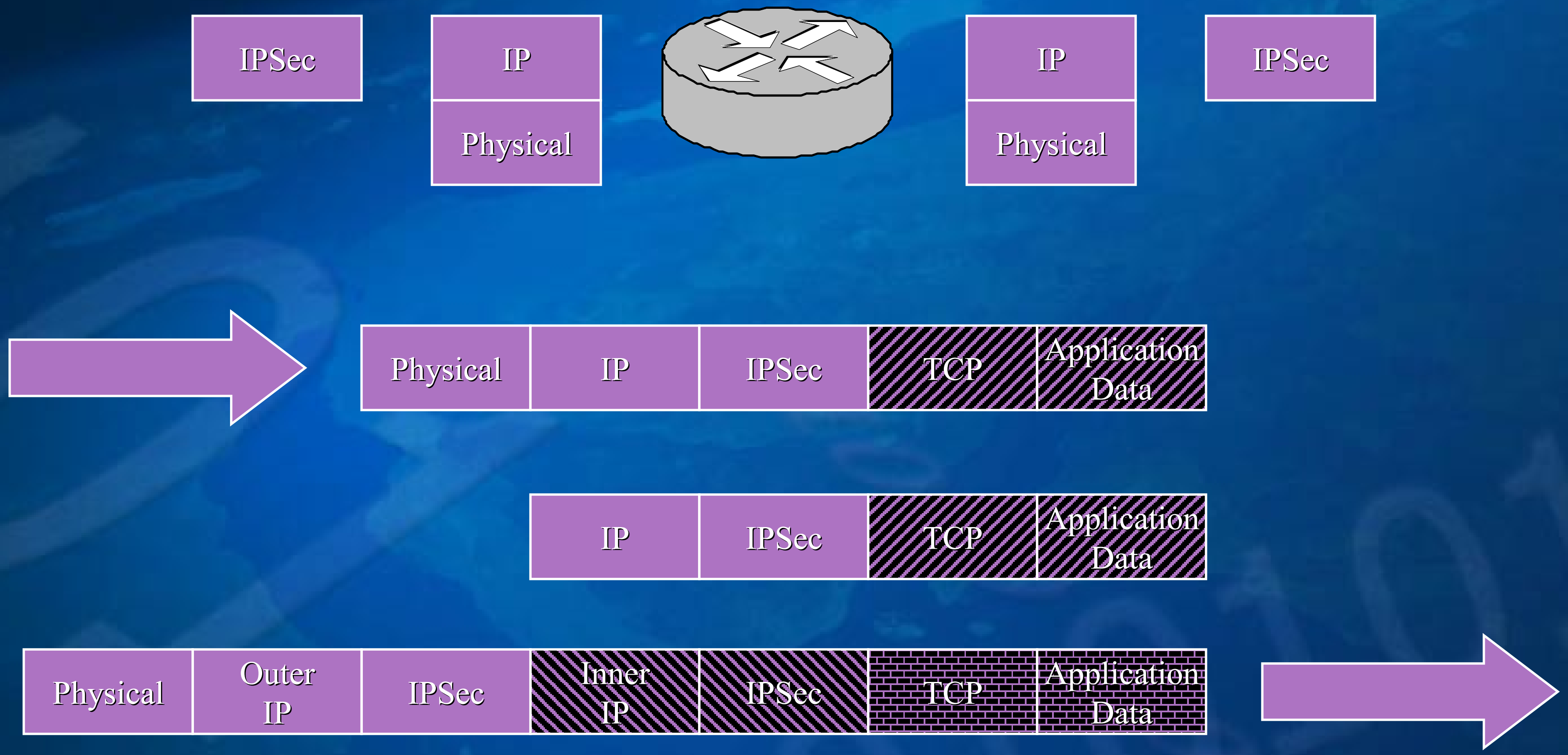
IP

Physical

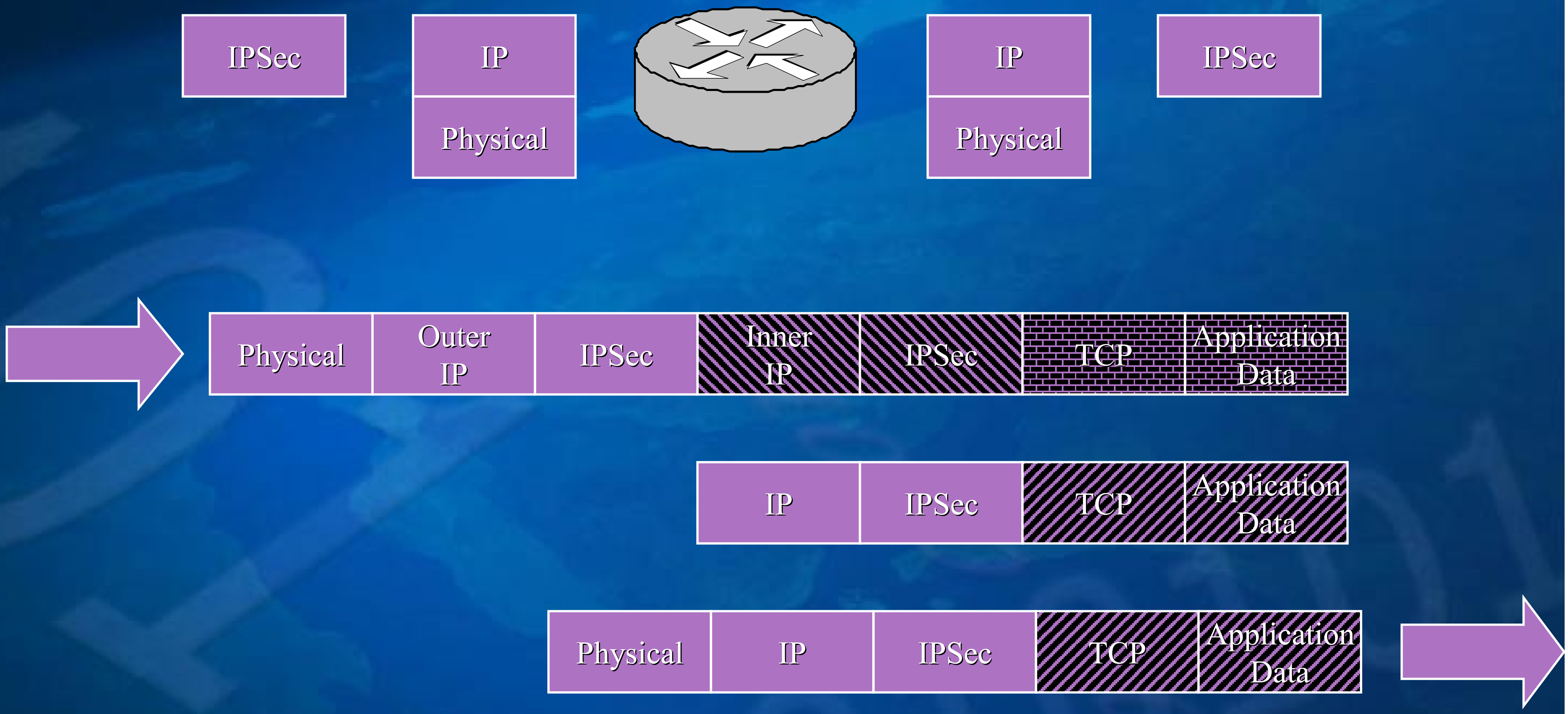
IPSec

Physical | IP | IPSec | TCP | Application Data

Sending in Tunnel Mode



Receiving in Tunnel Mode



IPSec

IP

Physical

IP

Physical

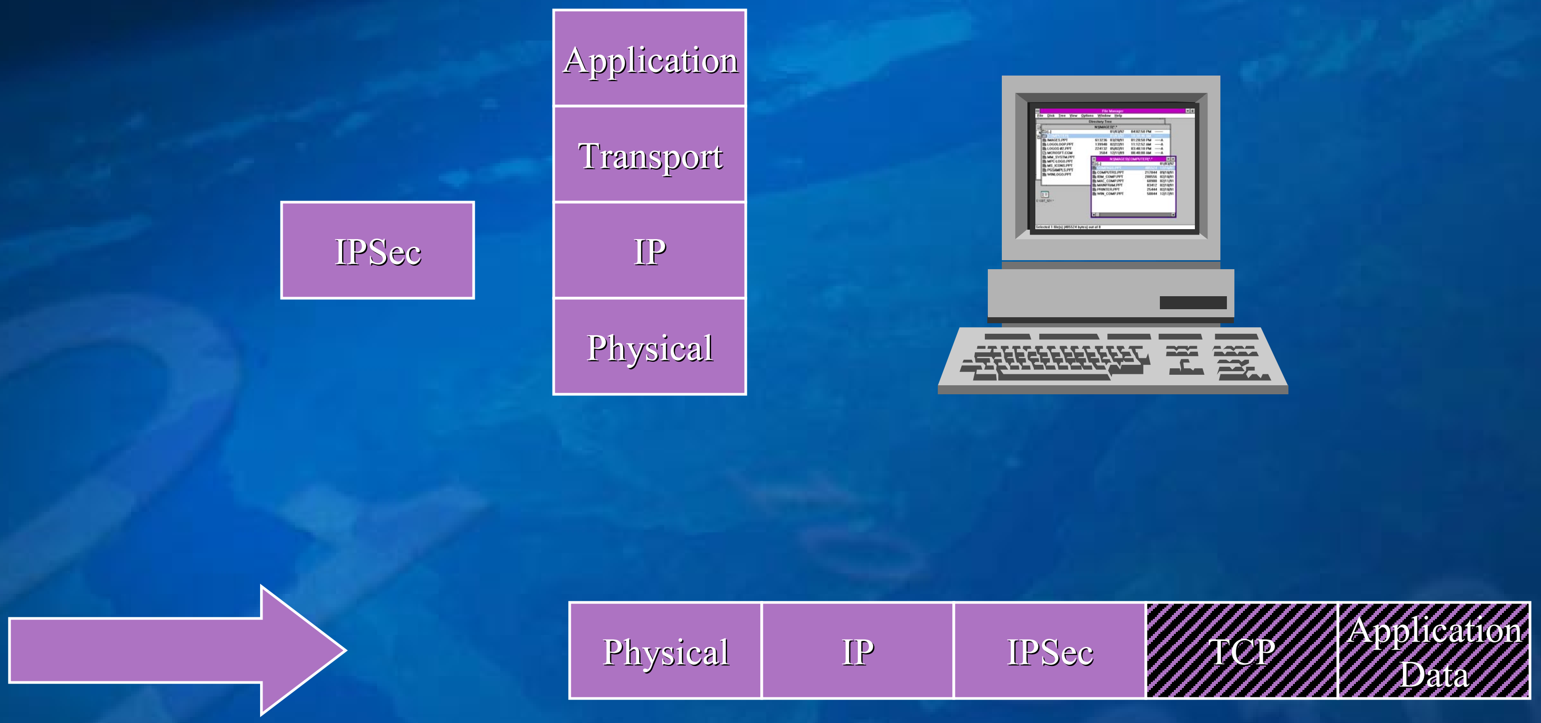
IPSec

Physical | Outer IP | IPSec | Inner IP | IPSec | TCP | Application Data

IP | IPSec | TCP | Application Data

Physical | IP | IPSec | TCP | Application Data

Receiving in Transport Mode



IPSec

Application

Transport

IP

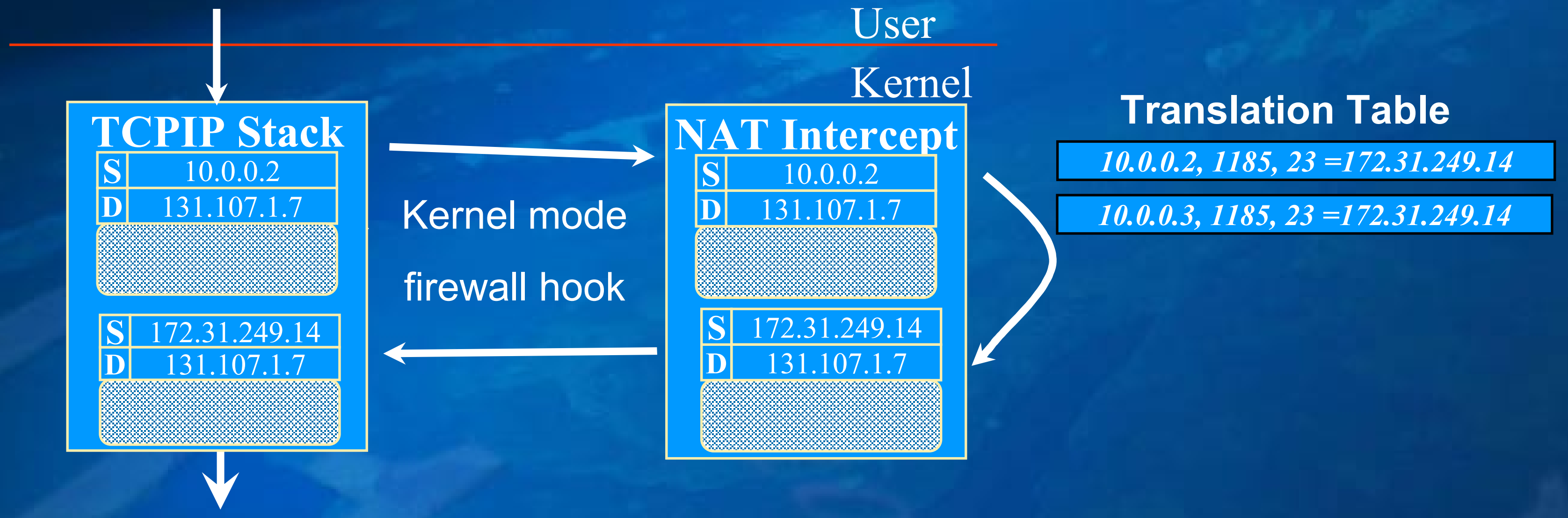
Physical

Physical | IP | IPSec | TCP | Application Data

What is Network Address Translation (NAT) ?

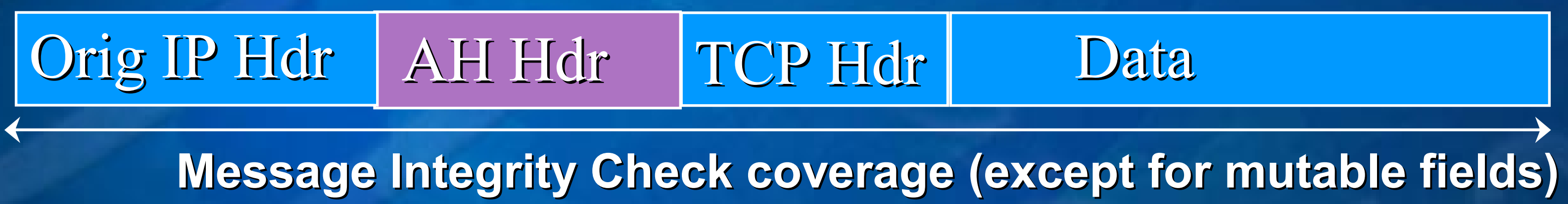
- Network Address Translation (NAT)
  - Dynamically modifies source address
  - Dynamically recomputes interior UDP/TCP checksums
- Port Address Translation (PAT)
  - Dynamically modifies TCP/UDP source address and port
  - Dynamically recomputes interior UDP/TCP checksums

NATs Rewrite Address/Port Pairs



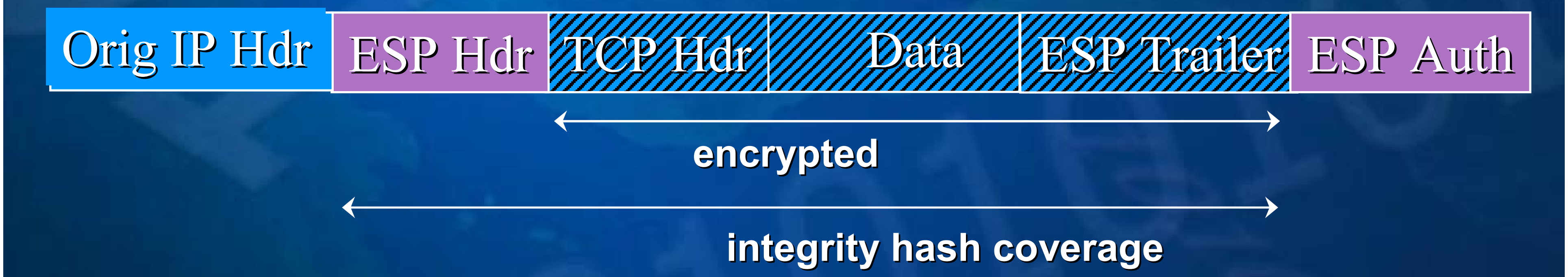
IPSEC AH and NAT

- Change in address or port will cause message integrity check to fail
  - Packet will be rejected by destination IPSEC
  - AH cannot be used with NAT or PAT devices


Orig IP Hdr
AH Hdr
TCP Hdr
Data
Message Integrity Check coverage (except for mutable fields)

IPSEC ESP and NAT

- Can change IP header in special cases only
  - Special TCP/UDP ignores pseudo header used in checksum calculation
- Port information encrypted!
- Can't change ESP header because integrity hash coverage



Message-based Protocols

Message-Based Protocols

- "Session" vs. "Message"
  - Synchronous vs. Asynchronous
- In message-based protocols, we cannot assume we have the luxury of being able to negotiate ciphersuites, parameter values, etc.
- In the common scenario, each message is a "fire-and-forget" communication
  - Each message has to contain enough information to allow the recipient to decrypt it.

Message-Based Protocols

- There are lots of message-based protocols
  - Examples: RPC, routing table updates
- The most common scenario to date, though, is e-mail
  - Digitally signed for sender authentication and integrity protection
  - Encrypted for confidentiality

S/MIME

Secure MIME
What is S/MIME?

- Secure Multipurpose Internet Mail Extensions
- Initially designed by RSA-led vendor consortium in 1995
- S/MIME messaging and S/MIME certificate handling are Internet RFC’s
  - Widely supported format for secure e-mail messages
  - Uses X.509v3 certificates

Scenario Assumptions

- Each participant has two public-private key pairs: one for signing messages and one for receiving encrypted messages from others
  - “Separation of duty” – separate keys (with separate controls) for separate uses
  - Encryption key archival/escrow/recovery
- For now, we assume key distribution isn’t a problem for participants
  - If I want to send you a message, I can obtain a copy of your encryption public key that I trust.
  - If you want to verify a message I signed, you can obtain a copy of my public signing key that you trust.

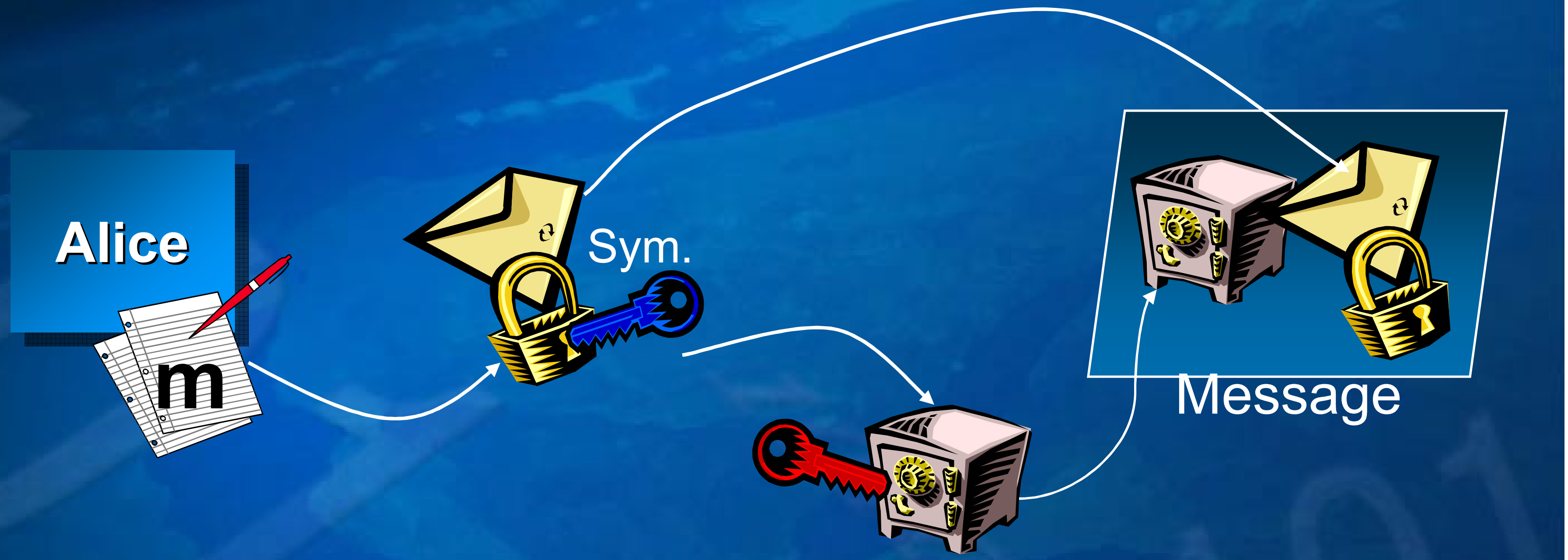
Encrypting Messages

- How do we want to encrypt messages?
- We have public keys for recipients, so we could repeatedly apply PK-encryption to portions of the message
  - Recall that we can only RSA-encrypt messages M with $|M| \leq |n|$
  - Plus, public key encryption is relatively slow, so we'd like to use it efficiently
- Idea: use PK to convey a random symmetric "session" key to recipients

Encrypting Messages

- We use symmetric encryption with randomly-generated session keys to encrypt message bodies
  - Since symmetric encryption is fast and messages may be arbitrarily large
- We use public-key encryption to encrypt the session keys to message recipients
- We send both encrypted message and session key as a unit to recipients…

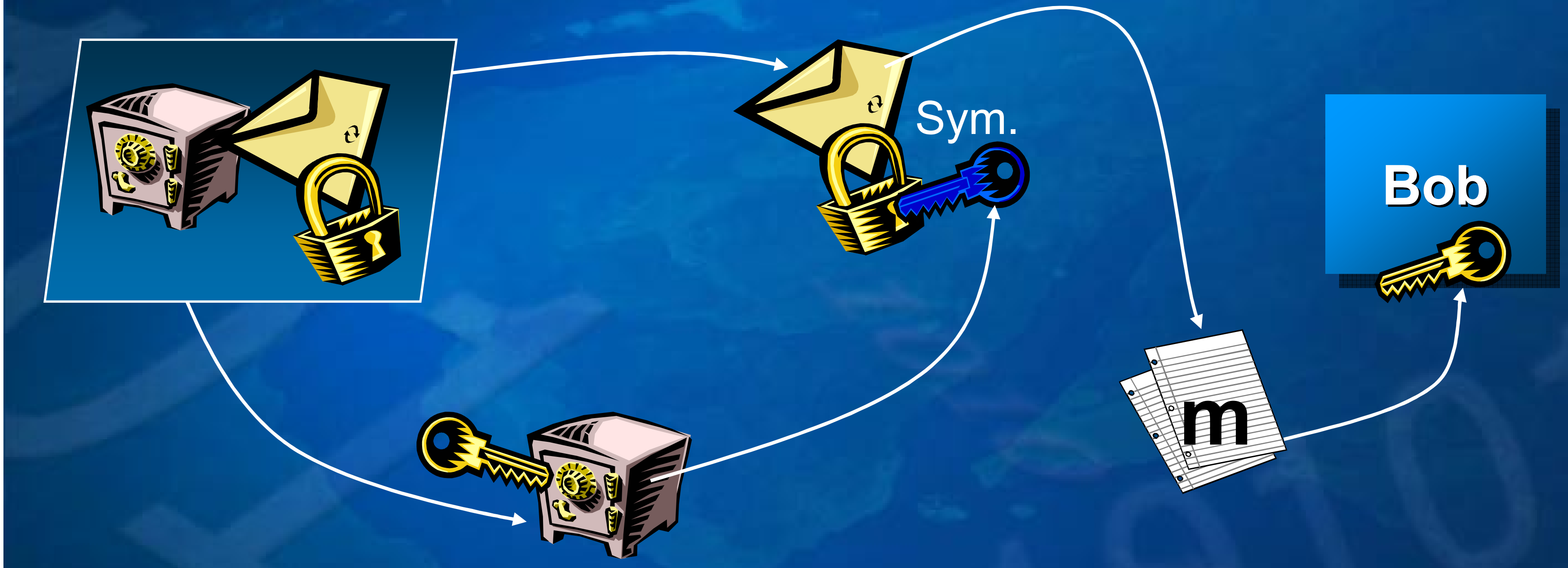
Message Encryption



Decrypting Messages

- Message decryption is just the reverse from encryption
- Recipients use their private encryption key to decrypt the session key for the message
- Recipients then use the session key to symmetrically decrypt the message body.

Message Decryption



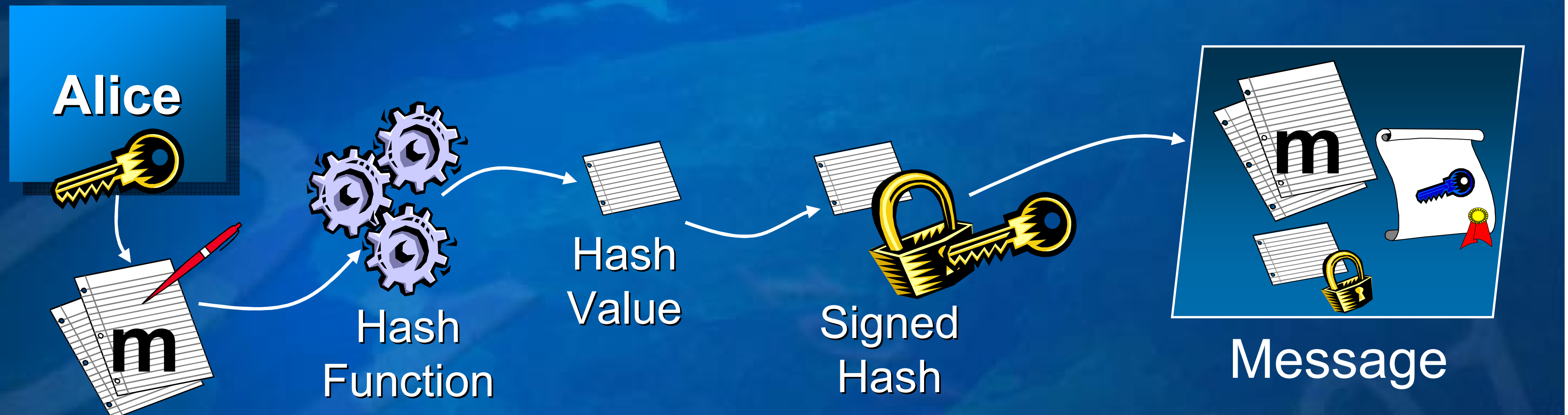
Signing Messages

- How do we want to sign messages?
- Each user has a signing key pair, but again we can only sign values that are at most the same size as our signing public key modulus
  - So we can’t sign the entire message directly, and repeated signing of parts of the message would open us up to attacks
- Idea: Sign a <u>hash</u> of the message

Signing Messages

- To sign a message, we first choose a cryptographic hash function H() to use with our signature algorithm
  - Normally defined as part of a signing ciphersuite
- We apply the hash function H to the exact sequence of bytes that forms our message (usually including header info)
- We sign the hash value
- We append the signed hash value to the message.

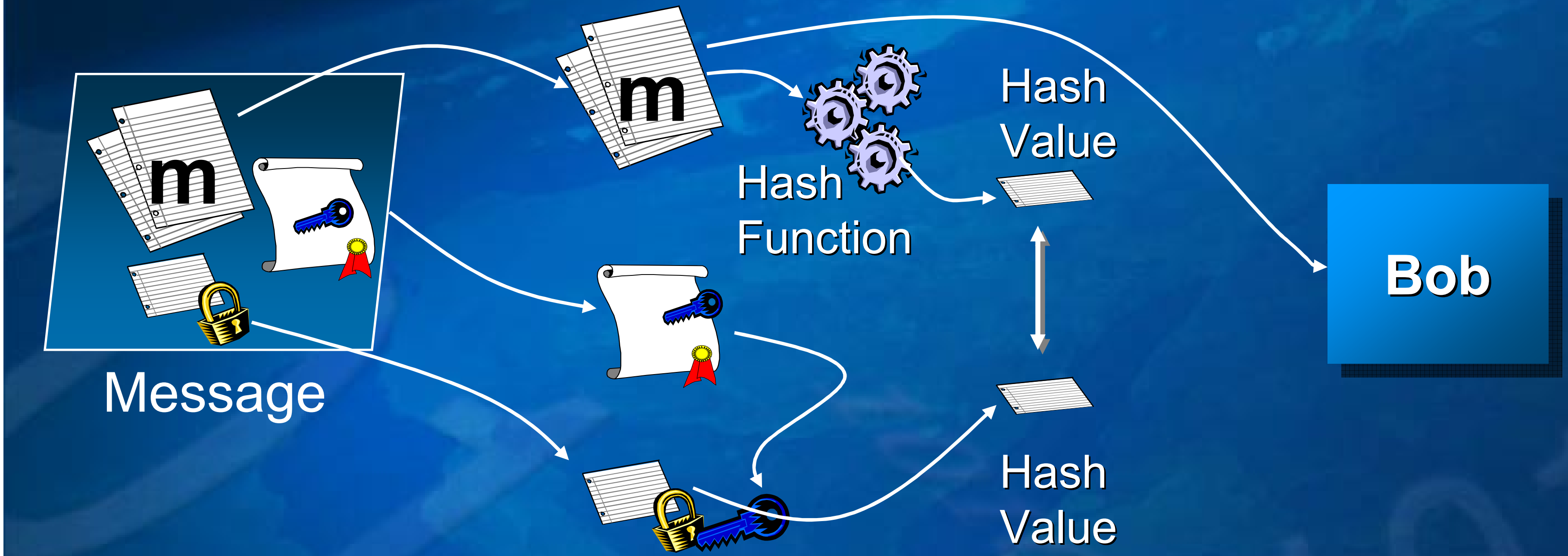
Digital Signatures
Provide Authentication and Integrity



Verifying Signatures

- To verify a signed message, the recipient has to do three things:
  1. Independently compute the hash value of the signed portion of the message
  2. Verify that the signature on the message came from the sender (by applying the sender’s public signing key)
     - This yields the hash value signed by the sender
  3. Compare the independently-computed hash value with the one the sender signed
- If the hash values are equal, then the message has not been modified since it was signed.

Verifying Signatures

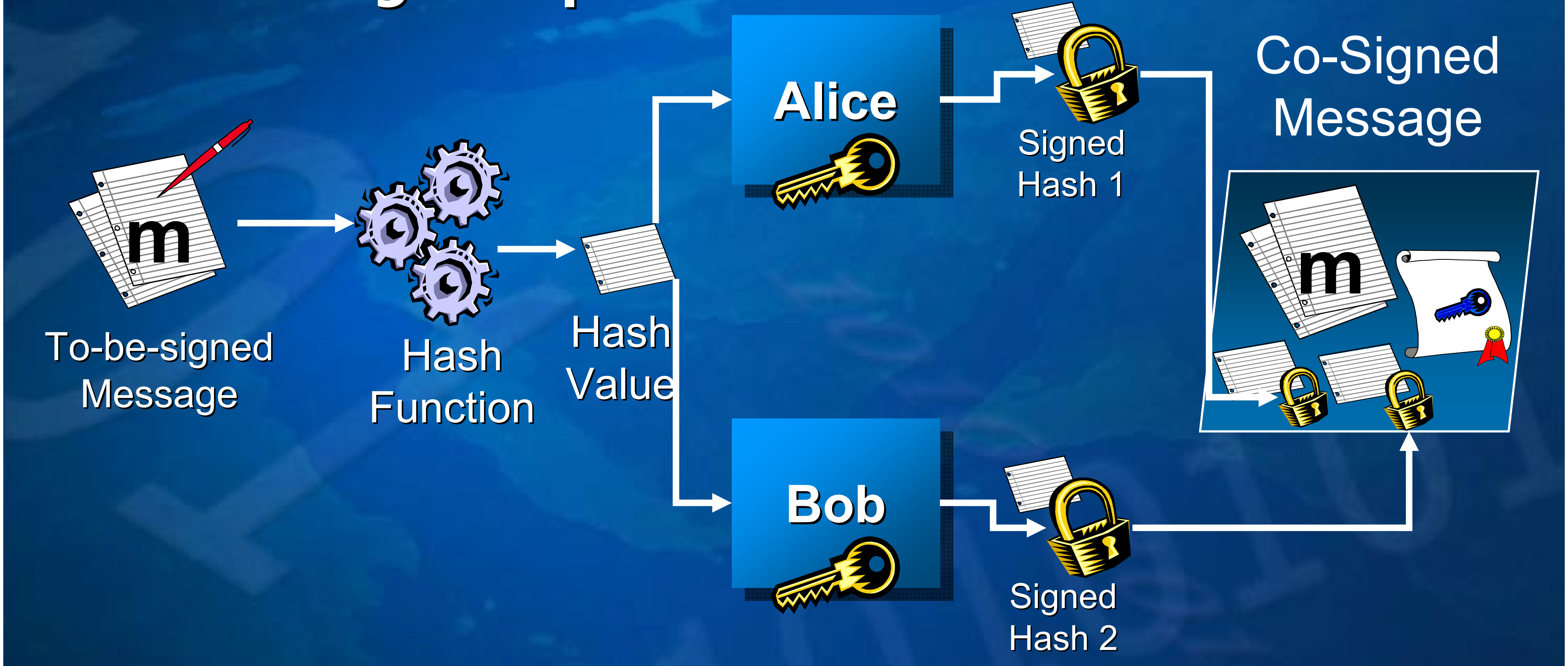


More Complex Signatures

- A single signer acknowledging understanding or commitment to different concepts or agreements within one document.
- Multiple signers signing unique content within the same document.
- Multiple signers “co-signing” the same content within the same document.
- Multiple signers, one signing content the other “counter-signing” the prior signature.

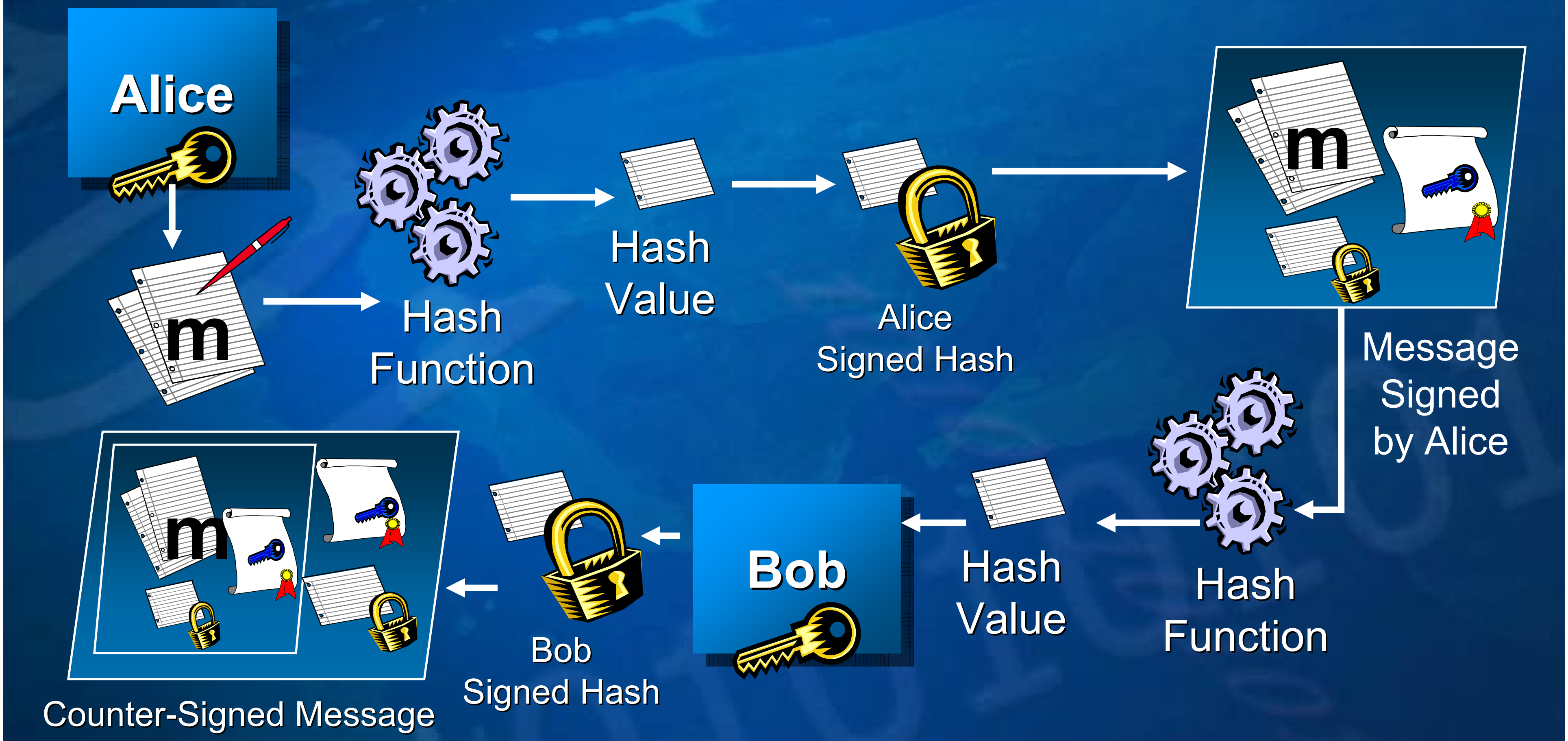
Co-Signing

- Alice and Bob want to sign the same message "in parallel"

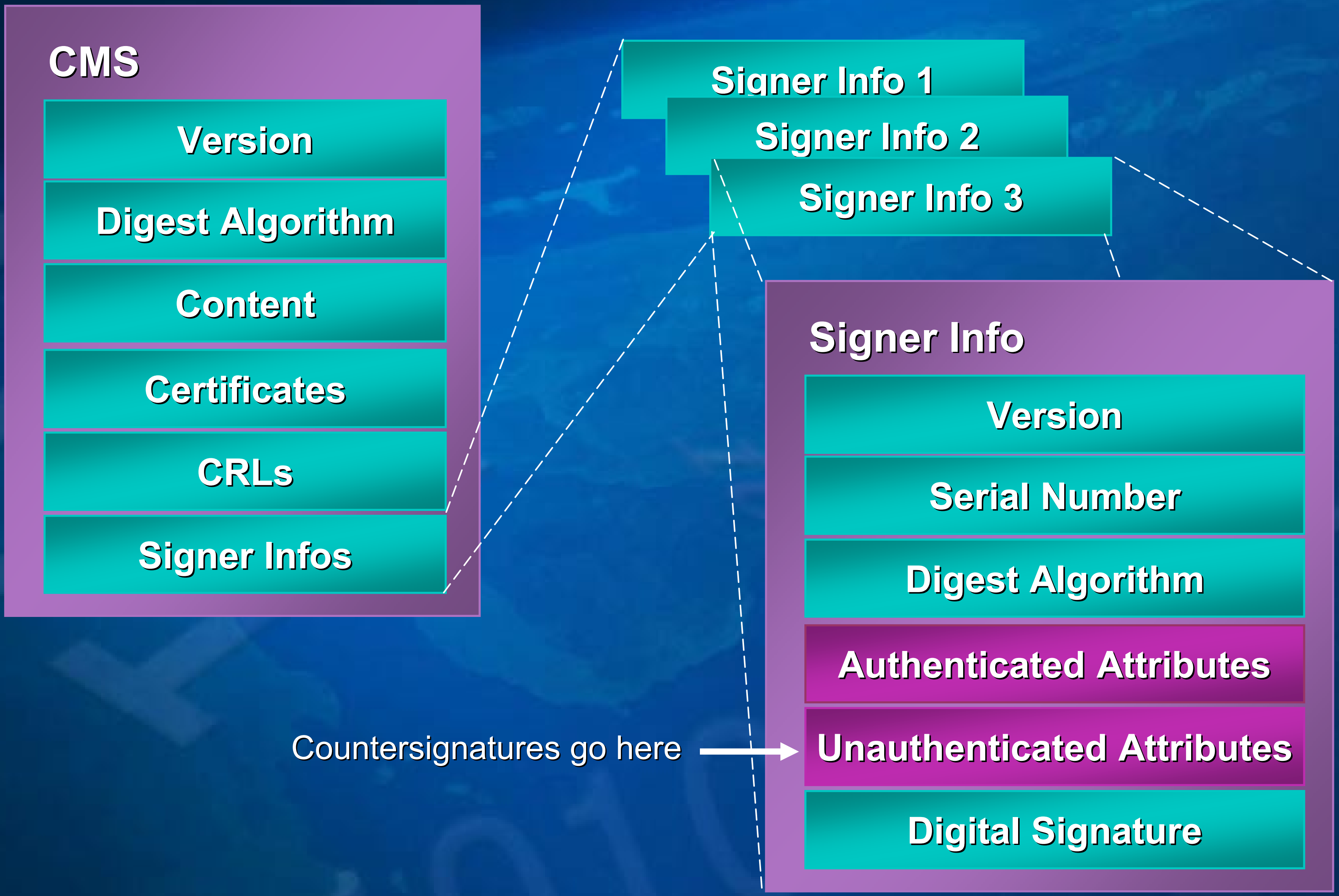


Counter-Signing

- Alice and Bob want to sign the same message “in series” (Alice first, then Bob)



PKCS #7/CMS Structure



CMS

- Version
- Digest Algorithm
- Content
- Certificates
- CRLs
- Signer Infos

Signer Info 1

Signer Info 2

Signer Info 3

Signer Info

- Version
- Serial Number
- Digest Algorithm
- Authenticated Attributes
- Unauthenticated Attributes
- Digital Signature

Countersignatures go here

Limitations of the CMS format

- The CMS standard only covers "wrapped" signatures
  - Signatures where the signed content is enclosed by the signature object
- Signing assumes you start with a bytestream that is completely immutable
  - This is the safest assumption, but sometimes it's overly conservative
  - Example: CR-LF rewriting and tab/whitespace conversions for text.

Message security for XML objects: XMLDSIG, XMLENC & WS-Security

What is XML?

```
<Address>
    <Street>1 Microsoft Way</Street>
    <City>Redmond</City>
    <State>WA</State>
    <ZipCode>98052</ZipCode>
</Address>
```

What is XML?

- XML is a W3C standard for describing "markup languages"
  - XML == "eXtensible Markup Language"
- Had its roots in SGML (of which HTML is an offshoot)
- Now, though, XML has really become a standard means of representing data structures in text.
  - "XML provides a text-based means to describe and apply a tree-based structure to information." -- Wikipedia

Securing XML

- As XML’s popularity grew, so did the need to secure XML objects (trees of XML elements)
- How should we sign & encrypt XML?
- One possibility: just treat an XML object as a byte sequence and use S/MIME
  - It’s just a sequence of characters, so we can Unicode encode that sequence, hash it, encrypt it and wrap it in S/MIME

Securing XML

- Using S/MIME works, but it has some drawbacks:
  1. The result of signing or encrypting an XML object is now some binary blob, not an XML object, so signing & encrypting this way doesn’t “play nice” with the XML ecosystem
  2. An XML object isn’t a piece of text – that text is just a representation of the object
     - There are many equivalent representations of an XML object
  3. There are semantically-neutral transforms allowed on XML representations that should not break signatures.

Signing & Encrypting XML

- Thus, there was a need to develop a standard for signing & encrypting XML objects
  - July 1999: work began on XMLDSIG, a standard for signing XML objects and representing signatures as XML
  - Summer 2000: work began on XMLENC, a standard for encrypting data and representing the ciphertext and associated key information as XML

XMLDSIG

The XMLDSIG Standard

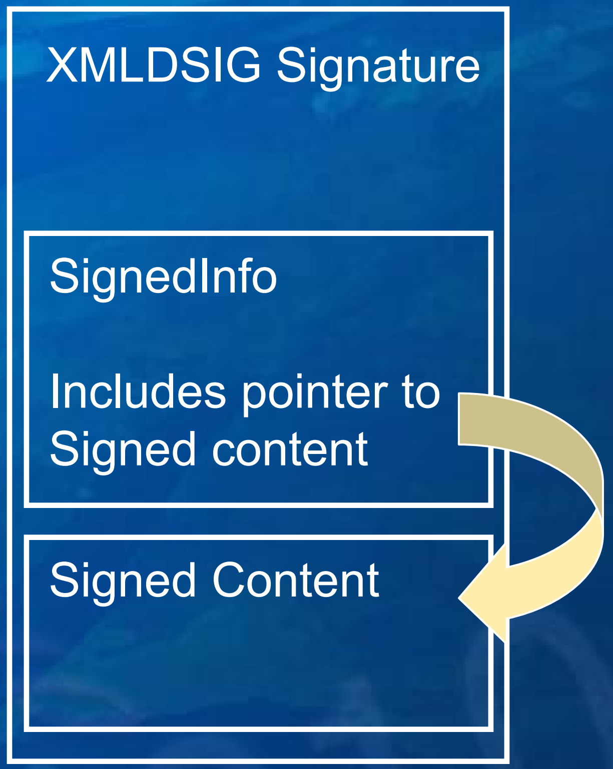
- XMLDSIG is an IETF/W3C joint standard for XML Digital Signatures
  - Signatures are represented as XML objects
  - Signed content may be XML documents, document fragments, or any binary stream
  - Baseline standard for further security work on XML Web Services (WS-Security)

Major Requirements and Key Features of XMLDSIG

- XMLDSIG supports three methods of signing an XML element
  - Wrapped, Detached and Embedded
- XMLDSIG signatures can be over an entire XML document or a fragment (sub-part) of a document
- XMLDSIG has to support the fact that an XML object might have multiple representations
  - Some modifications to the text must be allowed and not break the signature
- XMLDSIG has to support signatures over groups or collections of XML objects

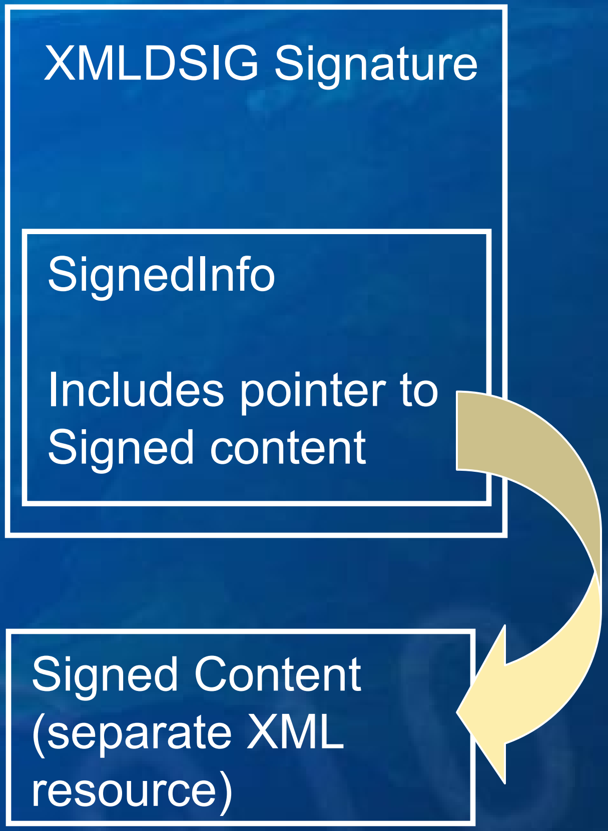
Wrapped Signatures

- Wrapped signatures include the signed content within the XMLDSIG structure
- Similar in format to a CMS (S/MIME) message
- Useful if the amount of to-be-signed data is small
  - Note: the signed content's schema is not preserved at top-level



Detached Signatures

- Detached signatures separate the signature from the signed content
  - Signature travels in a separate XML document
- Useful when you want to sign non-XML data
  - E.g. audio/visual data stream



XMLDSIG Signature

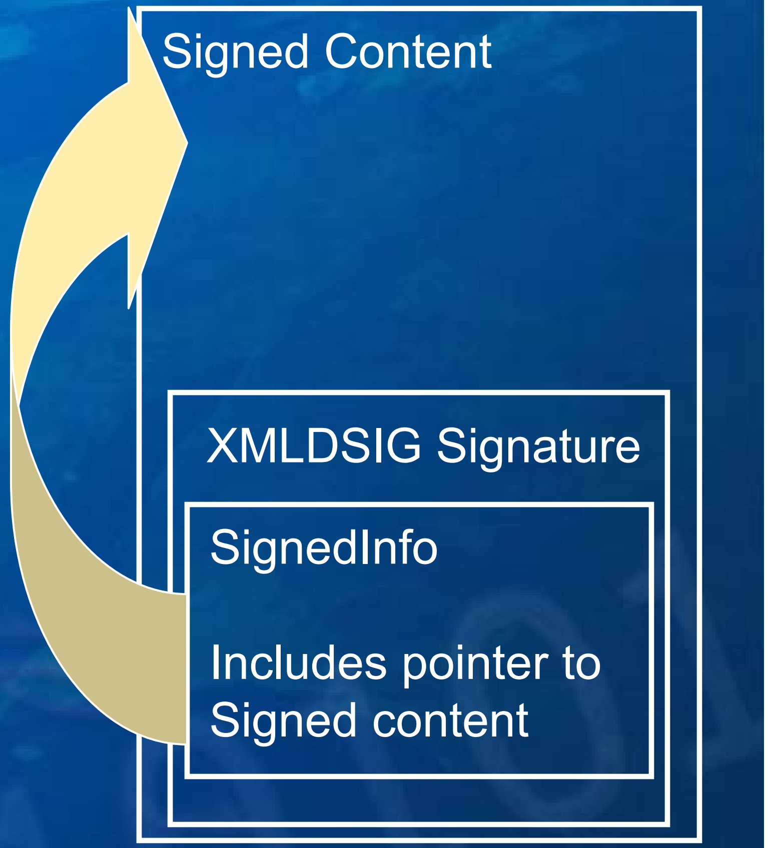
SignedInfo

Includes pointer to Signed content

Signed Content (separate XML resource)

Embedded Signatures

- New mechanism unique to XMLDSIG
- Standard way to embed an XMLDSIG signature within another XML document
- Signed document carries the signature inside itself



Signed Content

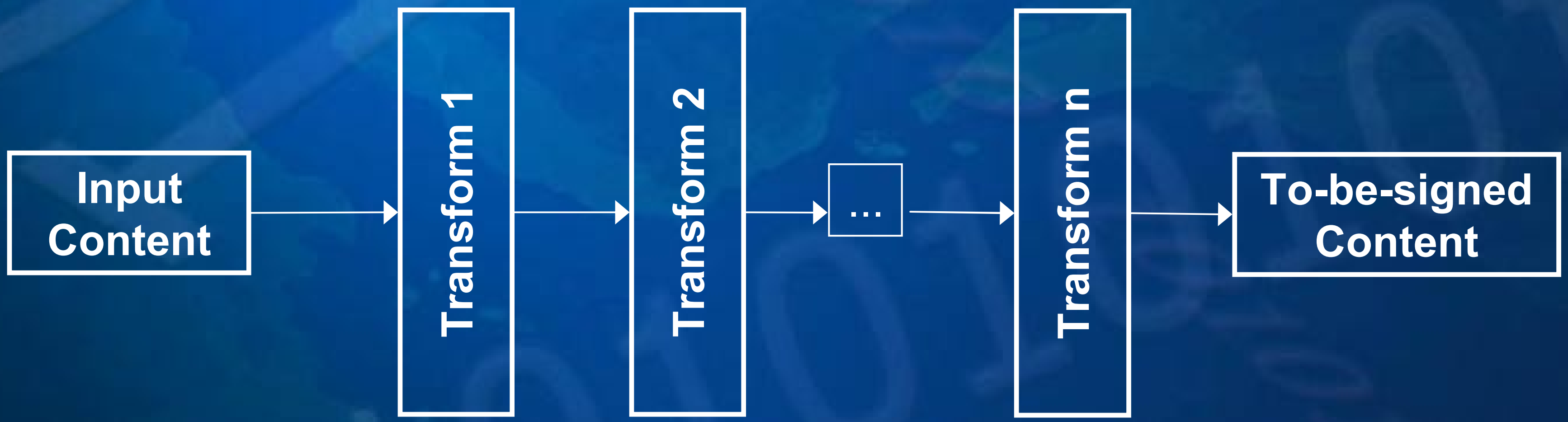
XMLDSIG Signature

SignedInfo

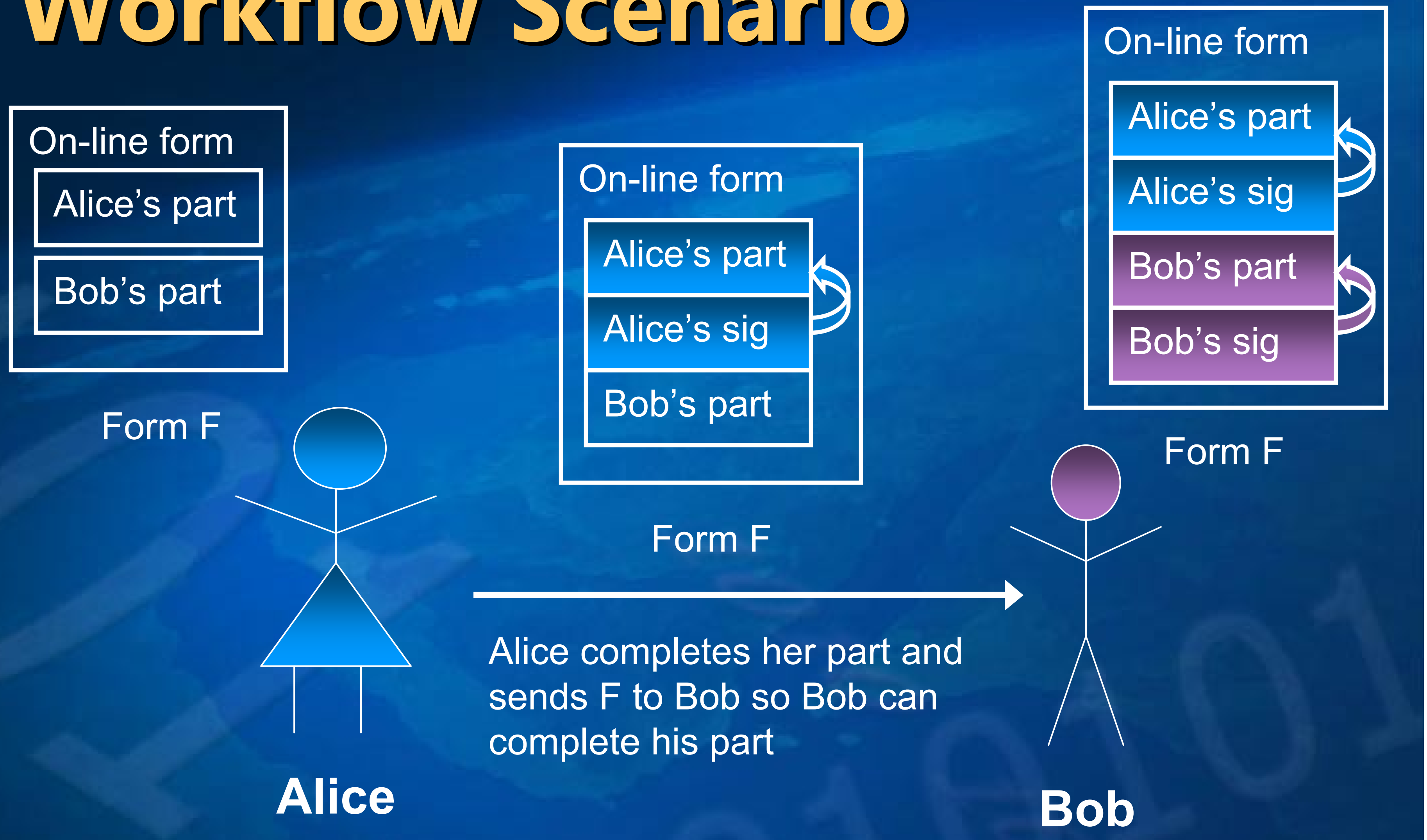
Includes pointer to Signed content

Signing Portions of Docs

- A key feature of XMLDSIG is its ability to sign selected portions of documents
  - Instead of hashing the entire document, identify & hash only those sections requiring protection
  - “Transform processing model”



Workflow Scenario



On-line form

Alice's part

Bob's part

Form F

On-line form

Alice's part

Alice's sig

Bob's part

Form F

On-line form

Alice's part

Alice's sig

Bob's part

Bob's sig

Form F

Alice

Alice completes her part and sends F to Bob so Bob can complete his part

Bob

Alice starts with a blank form

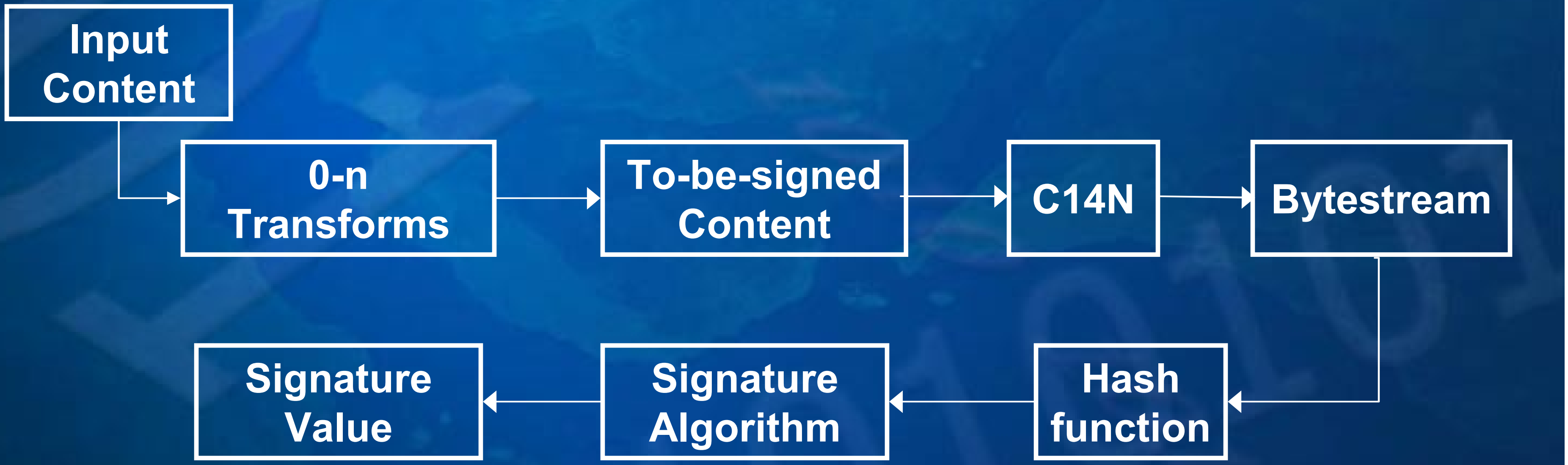
Bob completes his part and fills out the remainder of the form

Canonicalization (C14N)

- XMLDSIG introduced the notion of a "canonical form" for an XML object
  - C14N is an algorithm that converts an XML text representation into its canonical form bytestream.
  - All semantically-equivalent representations of an XML object have the same canonical form bytestream
    - That's the ideal case – in practice for various technical reasons we don't quite get there

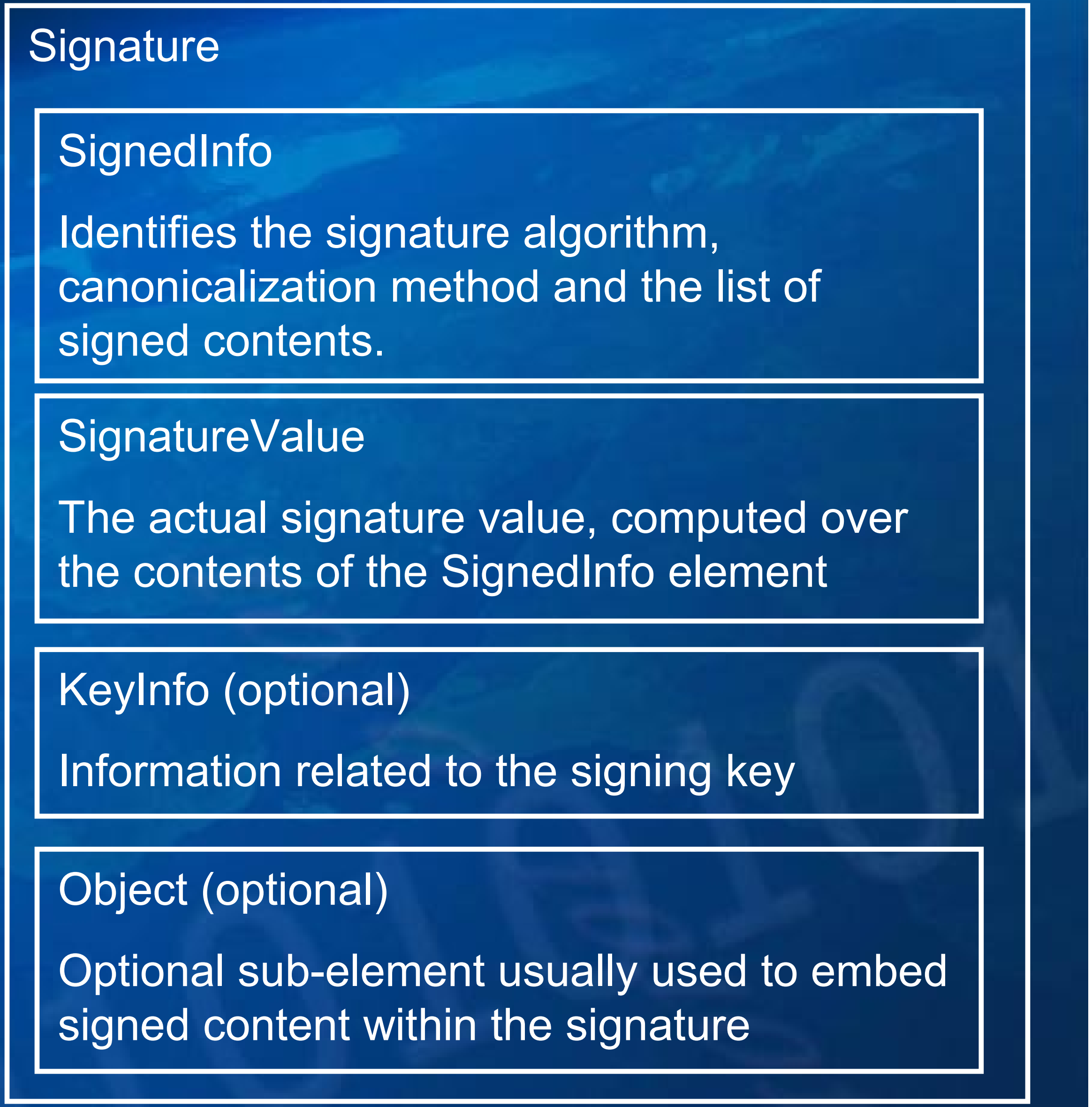
C14N and Signing

- In XMLDSIG, we compute the digital signature over the hash of the canonical form of whatever we want to sign



Structural Overview

- Top-level element is always a <Signature>
  - <SignedInfo> and <SignatureValue> are required sub-elements
  - <Keyinfo> and <Object> are optional



Signature

SignedInfo

Identifies the signature algorithm, canonicalization method and the list of signed contents.

SignatureValue

The actual signature value, computed over the contents of the SignedInfo element

KeyInfo (optional)

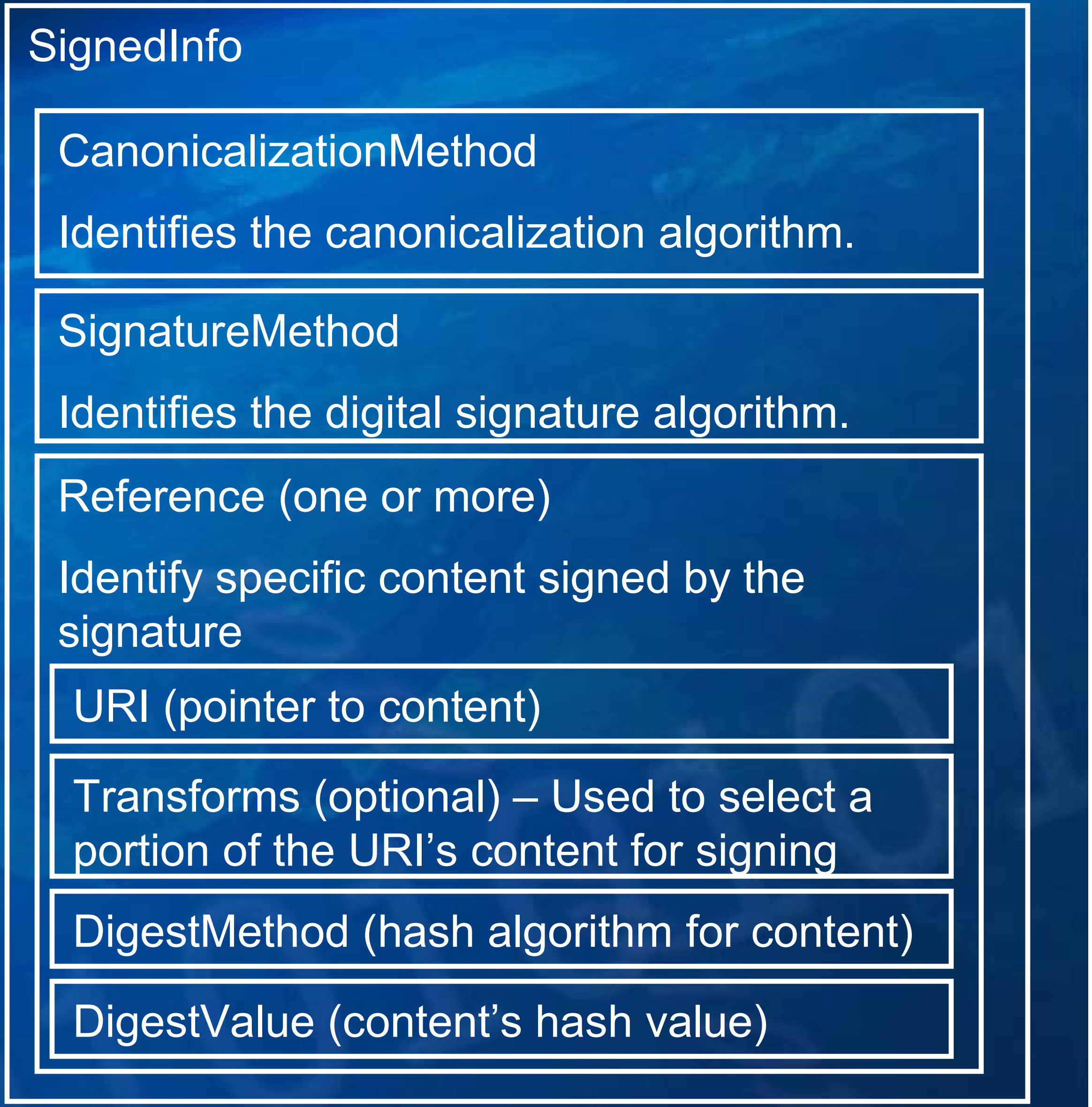
Information related to the signing key

Object (optional)

Optional sub-element usually used to embed signed content within the signature

SignedInfo Details

- The <SignedInfo> element contains a list <Reference> elements
- Each <Reference> element points to a piece of signed content
  - <SignedInfo> is a manifest listing all the contents signed by the signature



SignedInfo

CanonicalizationMethod

Identifies the canonicalization algorithm.

SignatureMethod

Identifies the digital signature algorithm.

Reference (one or more)

Identify specific content signed by the signature

URI (pointer to content)

Transforms (optional) – Used to select a portion of the URI’s content for signing

DigestMethod (hash algorithm for content)

DigestValue (content’s hash value)

Sample Signature

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
    20010315" />
    <SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-
    sha1"/>
    <Reference URI="http://www.farcaster.com/index.htm">
      <DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>XoaHIm+jLKnPocR7FX0678DUOqs=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>
    M5BhlrxPaOEYcCwSZ3WEDR6dfK5id/ef1JWK6OO5PEGHp9/JxrdA2xT5T
    Yr5egArZGdVURpMVGUeViWoeHcGAyMNG9Cmc/I56sYd/TSV/MjLgb/mxq
    +6Fh/HWtVhjHIG+AdL4lA+ZxxEi147QVVzgCl4+dvIZaGo7oAFneDKv0I
    =
  </SignatureValue>
</Signature>
```

XMLENC

The XMLENC Standard

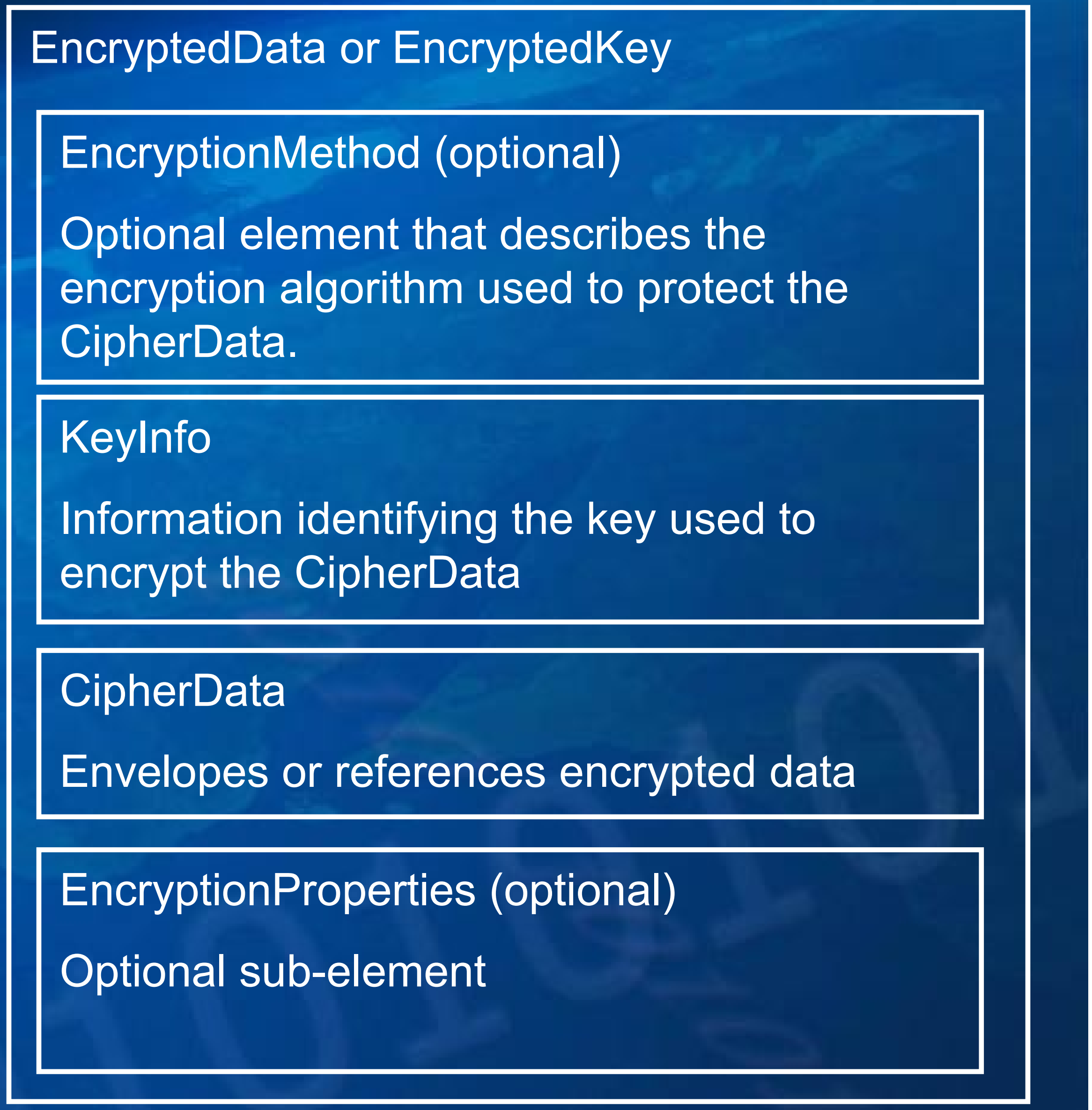
- XMLENC is a W3C Standard defining how to encrypt data and represent the result in XML
  - The data may be arbitrary data (including an XML document), an XML element, or XML element content.
  - The result of encrypting data is an XML Encryption element which contains or references the cipher data.

Key Features of XMLENC

- Wrapped or detached CipherData
  - Encrypted data may be enclosed within the metadata describing how it was encrypted, or sent separately
- EncryptedKey inside KeyInfo
  - Bulk data encryption keys wrapped in recipient public keys can be sent along with the data (a la S/MIME)
- Detached CipherData references use the same Transforms structure as XMLDSIG

Structural Overview

- Top-level element is either <EncryptedData> or <EncryptedKey>
- <EncryptedKey> has two additional properties over <EncryptedData>
  - <CipherData> always contains key material
  - An <EncryptedKey> may appear within an <EncryptedData>'s <KeyInfo> element.



EncryptedData or EncryptedKey

EncryptionMethod (optional)

Optional element that describes the encryption algorithm used to protect the CipherData.

KeyInfo

Information identifying the key used to encrypt the CipherData

CipherData

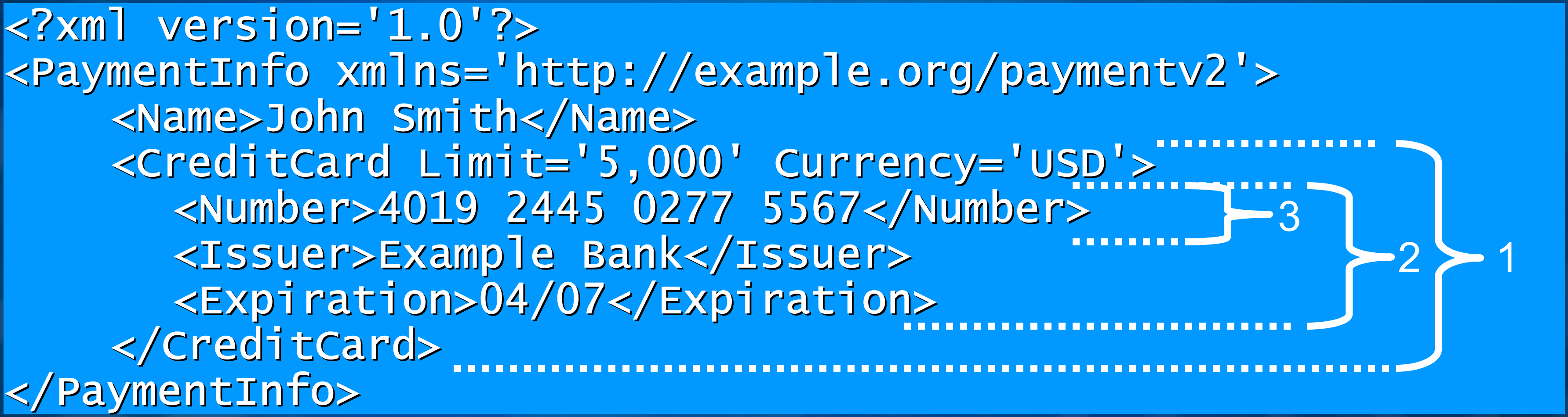
Envelopes or references encrypted data

EncryptionProperties (optional)

Optional sub-element

XMLENC Example

- Raw (unencrypted) XML: a simple payment structure with embedded credit card information



```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <CreditCard Limit='5,000' Currency='USD'>
      <Number>4019 2445 0277 5567</Number>
      <Issuer>Example Bank</Issuer>
      <Expiration>04/07</Expiration>
    </CreditCard>
</PaymentInfo>
```

XMLENC Example (1)

- Encrypting the entire <CreditCard> element including tag & attributes

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <EncryptedData
      Type='http://www.w3.org/2001/04/xmlenc#Element’
      xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
            <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
</PaymentInfo>
```

XMLENC Example (2)

- Encrypting the contents of <CreditCard> element

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <EncryptedData
      xmlns='http://www.w3.org/2001/04/xmlenc#‘
      Type='http://www.w3.org/2001/04/xmlenc#Content’>
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
  </CreditCard>
</PaymentInfo>
```

XMLENC Example (3)

- **Encrypting just the card number**

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>
      <EncryptedData
        xmlns='http://www.w3.org/2001/04/xmlenc#’
        Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>A23B45C56</CipherValue>
        </CipherData>
      </EncryptedData>
    </Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/07</Expiration>
  </CreditCard>
</PaymentInfo>
```

Web Services & WS-Security

Web Services in One Slide

- Software components accessible via standard “Web” protocols
  - Think of them as “remote procedure calls using SOAP/XML messages (over HTTP)”
- Available to any client that speaks XML, SOAP and the transport protocol
  - Platform independent components
- Enables Service-Oriented Architecture (SOA)-based application development
- Provides a general-purpose, composable protocol framework

Local Procedures

- Procedures create abstraction boundaries
  - Callers only care about inputs to & outputs from a procedure

```
public static float GetQuote(String symbol) {
  // implementation goes here
  // details are hidden from caller
}

public static void Main(String[] args) {
        float msftPrice = GetQuote(“MSFT”);
        Console.WriteLine("MSFT: {0:F2}",msftPrice);
}
```

```
C:\>test.exe
MSFT: 27.50
```

Quote Request Message

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/”
        xmlns:ns1="urn:xmethods-delayed-quotes"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        SOAP-
        ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/>
  <SOAP-ENV:Body>
   <ns1:getQuote>
    <symbol xsi:type="xsd:string">MSFT</symbol>
   </ns1:getQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Quote Response Message

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
     xmlns:ns1="urn:xmethods-delayed-quotes“
     xmlns:xsd=http://www.w3.org/2001/XMLSchema
     xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
     xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
     SOAP-
     ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encodin
     g/">
 <SOAP-ENV:Body>
  <ns1:getQuoteResponse>
   <Result xsi:type="xsd:float">27.50</Result>
  </ns1:getQuoteResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Security Requirements

- Message-level security
  - Confidentiality, integrity and authentication for every SOAP request and response
  - Web services are asynchronous – no "channel"
- Interoperable
  - People, systems, applications, and services
  - Heterogeneous environments
- Can be composed with other SOAP protocol features
  - Ex: reliable messaging, transactions
- Decentralized and dynamic
  - Arbitrary network topology with no central authority
  - Assume policies change and evolve over time
  - Dynamic authorization model

WS-Security

- Defines a framework for building security protocols
  - Integrity
  - Confidentiality
  - Propagation of security tokens
    - Authorization credentials
- Framework designed for end-to-end security of SOAP messages
  - From initial sender, through 0-n intermediaries to ultimate receiver

What are security tokens?

- Represent claims about identity, capabilities, privileges

Username Token

X.509 Certificate

Kerberos Ticket

Protecting messages

- Parts of a message can be signed to ensure integrity
- Parts of a message can be encrypted to ensure confidentiality
- Underlying technologies support pluggable algorithms
  - Encryption, Digest, Signature, Canonicalization, Transforms

```
<s:Envelope
  xmlns:s='http://www.w3.org/2003/05/soap-envelope'
  xmlns:wsu='http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd'
  xmlns:ws='http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-secext-1.0.xsd'
  xmlns:ds='http://www.w3.org/2000/09/xmldsig#' >
 <s:Header>
  <ws:Security s:mustUnderstand='true' >
   <ws:BinarySecurityToken wsu:Id='Me'
  ValueType=‘http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-x509-token-profile-1.0#X509v3'
  EncodingType=‘http://dosc.oasis-open.org/wss/2004/01/oasis-
200401-wss-soap-message-security-1.0#Base64Binary' >
   MeIIZFgea4FGiu5cvWEklO8pl...
   </ws:BinarySecurityToken>
   . . .
```

My security token

```
   . . .
   <ds:Signature>
    <ds:SignedInfo>
     <ds:CanonicalizationMethod
 Algorithm='http://www.w3.org/2001/10/xml-exc-c14n#' />
     <ds:SignatureMethod
 Algorithm='http://www.w3.org/2000/09/xmldsig#rsa-sha1' />
     <ds:Reference URI='#Body' >
       <ds:DigestMethod
 Algorithm='http://www.w3.org/2000/09/xmldsig#sha1' />
       <ds:DigestValue>uJhGtef54ed91iKLoA...</ds:DigestValue>
     </ds:Reference>
    </ds:SignedInfo>

<ds:SignatureValue>FR8yaKmNDePQ7E3Hj...</ds:SignatureValue>
    . . .
```

Reference to data I want to protect

Digest of data I want to protect

Signature over ds:SignedInfo element

```
    . . .
    <ds:KeyInfo>
     <ws:SecurityTokenReference>
      <ws:Reference URI='#Me‘
  ValueType=' http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-x509-token-profile-1.0#X509v3' />
     </ws:SecurityTokenReference>
    </ds:KeyInfo>
   </ds:Signature>
  </ws:Security>
  . . .
 </s:Header>
 <s:Body wsu:Id='Body' >
  . . .
 </s:Body>
</s:Envelope>
```

Reference to certificate that can be used to verify signature

Signed data

Confidentiality example (Sender)

- I want to send a SOAP message and ensure that only you can read the content of the body
  - I generate a symmetric key
  - I encrypt that key using your public key
  - I encrypt the content of the body using the symmetric key
  - I include both the encrypted data and encrypted key in the message

```
<s:Envelope
  xmlns:s='http://www.w3.org/2003/05/soap-envelope'
  xmlns:wsu='http://docs.oasis-open.org/wss/2004/01/oasis-
  200401-wss-wssecurity-utility-1.0.xsd'
  xmlns:ws='http://docs.oasis-open.org/wss/2004/01/oasis-
  200401-wss-wssecurity-secext-1.0.xsd'
  xmlns:ds='http://www.w3.org/2000/09/xmldsig#'
  xmlns:xe='http://www.w3.org/2001/04/xmlenc#' >
 <s:Header>
  <ws:Security s:mustUnderstand='true' >
  . . .
```

```
   . . .
   <xe:EncryptedKey Id='Sym' >
    <xe:EncryptionMethod
 Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p'
/>
    <ds:KeyInfo>
     <ws:SecurityTokenReference>
      <ws:KeyIdentifier>
      aKKuvtdlAnUm+I6+ZTDrUA==
      </ws:KeyIdentifier>
     </ws:SecurityTokenReference>
    </ds:KeyInfo>
    <xe:CipherData>
      <xe:CipherValue>bvDfEg6Sh7GbCvDiAl</xe:CipherValue>
    </xe:CipherData>
    <xe:ReferenceList>
     <xe:DataReference URI='#EncBody' />
    </xe:ReferenceList>
   <xe:EncryptedKey>
  </ws:Security>
  . . .
 . . .
 </s:Header>
 <s:Body>
  <xe:EncryptedData Id='EncBody'
Type='http://www.w3.org/2001/04/xmlenc#Element' >
   <xe:EncryptionMethod
 Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc' />
   <ds:KeyInfo>
    <ws:SecurityTokenReference>
     <ws:Reference URI=‘#Sym’ />
    </ws:SecurityTokenReference>
   </ds:KeyInfo>
   <xe:CipherData>
    <xe:CipherValue>
    ABfg5eFdiKmNeQlPsDFoMNb...
    </xe:CipherValue>
   </xe:CipherData>
  </xe:EncryptedData>
 </s:Body>
</s:Envelope>
```

WS-Trust
(if we have time)

Authorization Model

- Web Services need mechanisms for conveying authorization information from client to server
  - "Is the client authorized to make this type of request and receive the results?"
- Use <u>security tokens</u> to convey authorizations
  - Capabilities-based model (sender proves he has the right to make the request)
  - Tokens contain <u>claims</u> that state properties
    - Ex: identity, age, state of residence
- Servers need a way to publish their authorization policies
  - "Who is allowed to call this web service?"
  - Policy describes required claims (and semantics)

Security token example

- Alice's X.509 certificate is a security token
  - Allows a message to claim to be from Alice
- Proof of claim is based on Alice's private key
  - Signing part of the message with her private key proves that she knows the key and is therefore Alice

WS-Trust

- Defines how to broker trust relationships
  - Some trust relationship has to exist *a priori* between the two parties
- Defines how to exchange security tokens
- Defined as an interface specification for a *Security Token Service*
  - STS = Token issuer

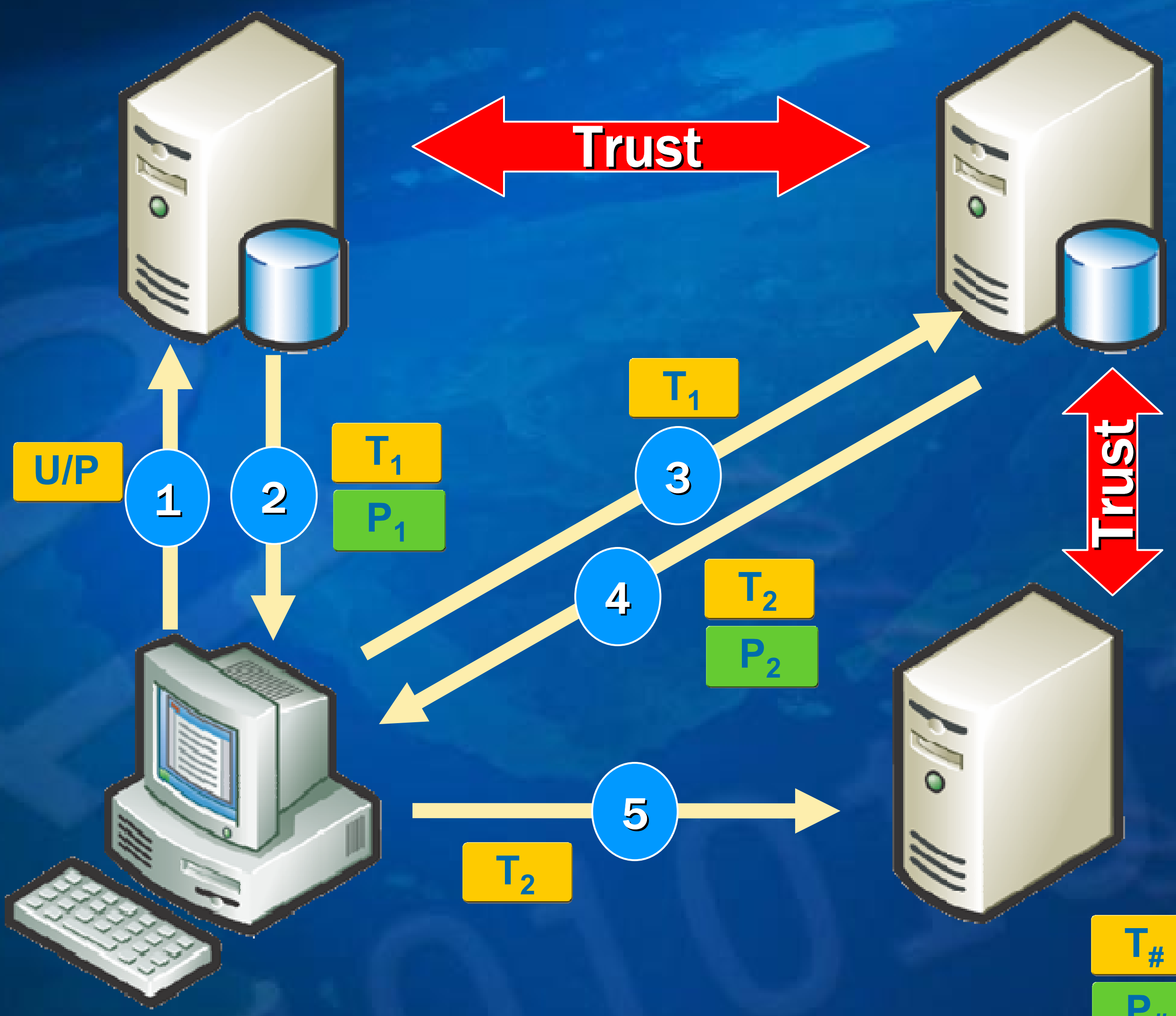
Common Patterns

- Issuance
  - Exchanging one set of credentials (optionally null) for another
- Renewal
  - Renewing previously issued tokens
- Validation
  - Verifying tokens and signatures using a service
- Cancellation/Revocation
  - Cancelling a previously issued token
- Challenges/Negotiations
  - How to have secure multi-leg challenges and negotiations prior to token issuance

Example

- I want to have a secure conversation with you
- I ask the trust service for a token to allow me to talk to you
- The trust service sends me a token containing two copies of a secret key
  - One encrypted for me
  - One encrypted for you
- The former is a “proof token”
  - I can use the secret key in it to respond to a challenge you give me

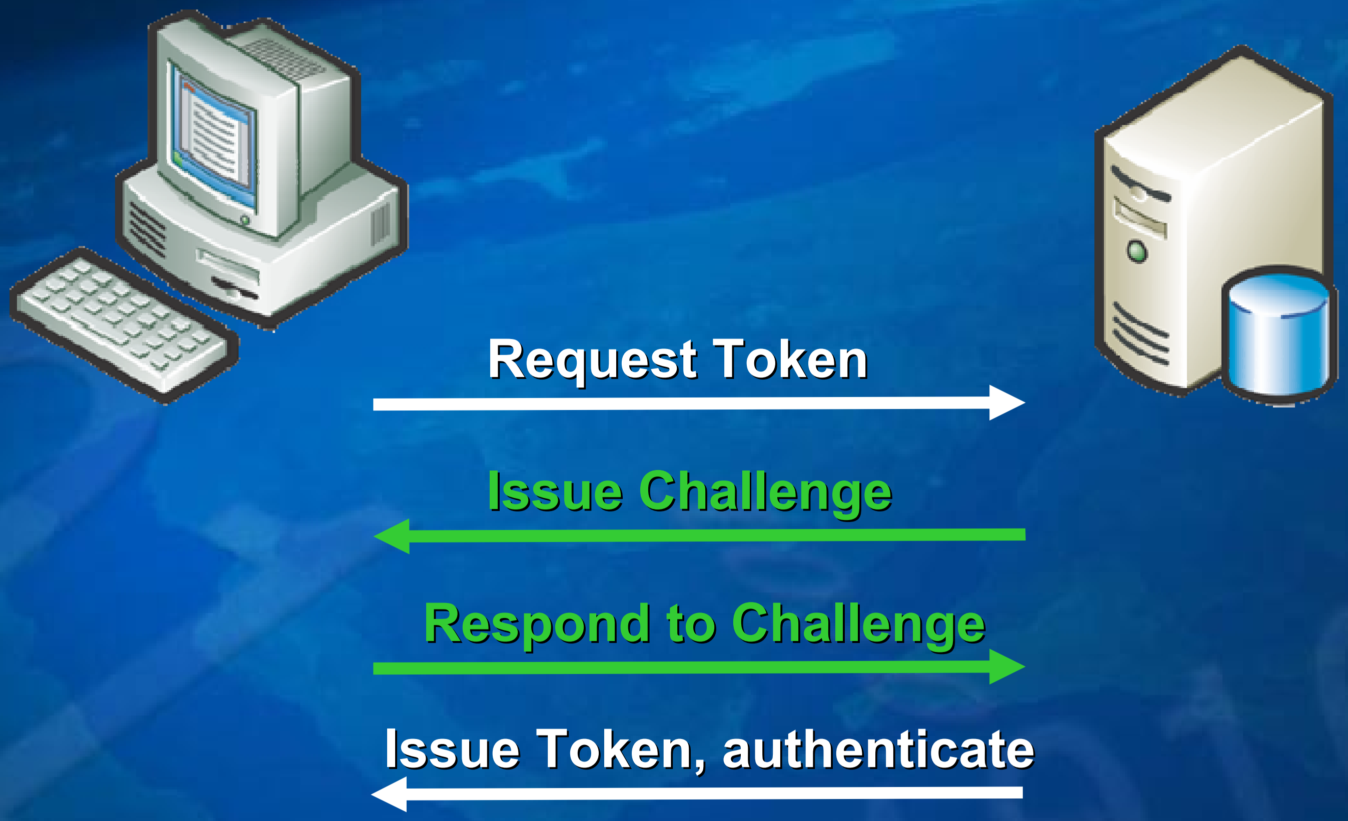
Example



$T_\#$ Security token

$P_\#$ Proof token

Challenges



Request Token

Issue Challenge

Respond to Challenge

Issue Token, authenticate

Getting Tokens

- A RequestSecurityToken message is sent to the trust service
- It responds with a RequestSecurityTokenResponse
  - Contains required security token and associated metadata/attributes/etc.
- Various bindings defined
  - A binding defines wsa:Action values and wst:RequestType values
    - E.g. Message types associated with the "Issue" action

Other token characteristics

- Requester can specify various required characteristics of the security token
  - Key type, size
  - Whether token is forwardable, delegateable etc.
- Trust service can then indicate those characteristics in the response