Vérification des Protocoles Cryptographiques et de leurs Implémentations

Cédric Fournet
http://research.microsoft.com/~fournet

+ Martin Abadi, Pedro Adao, Karthik Bhargavan, Bruno Blanchet, Ricardo Corin, Andrew D. Gordon, Tamara Rezk, Stephen Tse
Vérification des Protocoles Cryptographiques et de leurs Implémentations

1. Modélisation des protocoles en pi calcul
2. Comment vérifier l’usage des protocoles? applications aux services Web (outils, demo)
3. Comment vérifier leurs implémentations? (demo)
4. Cryptographie formelle/concrete
Context:

Verifying Cryptographic Protocols

What’s so difficult about protocols?
Cryptographic protocols go wrong

- Historically, one keeps finding simple attacks against protocols
  - even carefully-written, widely-deployed protocols, even a long time after their design & deployment
  - simple = no need to break cryptographic primitives

- Why is it so difficult?
  - concurrency + distribution + cryptography
    - Little control on the runtime environment
  - active attackers
    - Hard to test
  - implicit assumptions and goals
    - Authenticity, secrecy
In *Using encryption for authentication in large networks of computers (CACM 1978)*, Needham and Schroeder didn’t just initiate a field that led to widely deployed protocols like Kerberos, SSL, SSH, IPSec, etc.

They threw down a gauntlet.

“Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation.

The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area.”
The Needham-Schroeder public-key authentication protocol (CACM 1978)

Principal A initiates a session with principal B
S is a trusted server returning public-key certificates eg \{ | A,KA | \}_{KS^{-1}}
NA,NB serve as nonces to prove freshness of messages 6 and 7
Assuming A knows KB and B knows KA, we get the core protocol:

\[
\begin{align*}
\{ | \text{msg3}(A, NA) | \}_\text{KB} \\
\{ | \text{msg6}(NA, NB) | \}_\text{KA} \\
\{ | \text{msg7}(NB) | \}_\text{KB}
\end{align*}
\]

More precisely, the goals of the protocol are:
- After receiving message 6, A believes NA, NB shared just with B
- After receiving message 7, B believes NA, NB shared just with A

If these goals are met, A and B can subsequently rely on keys derived from NA, NB to efficiently secure subsequent messages.
A certified user M can play a man-in-the-middle attack (Lowe 1995)

This run shows a certified user M can violate the protocol goals:
- After receiving message 6, A believes NA,NB shared just with M
- After receiving message 7, B believes NA,NB shared just with A

(Writing in the 70s, Needham and Schroeder assumed certified users would not misbehave; we know now they do.)
Informal methods

Informal lists of prudent practices enumerate common patterns in the extensive record of flawed protocols, and formulate positive advice for avoiding each pattern.

(eg Abadi and Needham 1994, Anderson and Needham 1995)

“**Principle 1:**
Every message should say what it means: the interpretation of the message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content — though if there is a suitable formalism available that is good too.”

Abadi and Needham  *Prudent engineering practice for cryptographic protocols* 1994

For instance, Lowe’s famous fix of the Needham-Schroeder PK protocol makes explicit that message 6, \{\|NA,B,NB\|\}KA, is sent by B, who is not mentioned in the original version of the message.
Formal methods

- Dolev and Yao first formalize N&S problem in early 80s
  - They use algebraic, idealized cryptography
    - Shared key decryption: decrypt({M|K,K} = M
    - Public key decryption: decrypt(|M|K,A+,A-) = M
  - Their work is now widely recognised, but at the time few proof techniques, and little applied

- In 1987, Burrows, Abadi and Needham (BAN) also propose a systematic rule-based logic for reasoning about protocols
A brief history: 1978—2007

1978: N&S propose authentication protocols for “large networks of computers”
1981: Denning and Sacco find attack on N&S symmetric key protocol
1983: Dolev and Yao first formalize secrecy properties of NS threat model using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
1994: Hickman, Elgamal invent SSL; holes in v1, v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs

We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.

Needham and Schroeder CACM (1978)
2007: mission accomplished

- Authentication and secrecy properties for basic crypto protocols have been formalized and thoroughly studied

- After intense effort on symbolic reasoning, several techniques and tools are available for automatically proving these properties
  - e.g. Athena, TAPS, ProVerif, FDR, AVISPA, etc

- We can now automatically verify most security properties for detailed models of crypto protocols
  - e.g. IPSEC, Kerberos, Web Services, Infocard

- Ongoing work
  - Fancy protocols and properties
  - Relation between Dolev-Yao abstractions and concrete crypto
2007: mission accomplished?

- Best practice: apply formal methods and tools throughout the protocol design & review process

- Not so easy
  - Specifying a protocol is a lot of work
  - Most practitioners don’t understand formal models

- Protocols go wrong because...
  - they are logically flawed, or
  - they are used wrongly, or
  - they are wrongly implemented

- Two troublesome questions
  1. How to relate crypto protocols to application security?
  2. How to relate formal models to executable code?
Cryptographic Protocols and Application Security

How do you intend to use this protocol?
Application to Web Services security
Application vs protocol security

- Protocols only contribute to high-level application security
  - “Except for emergencies, only my doctor may access my medical record”

- Controlling the usage of protocols is difficult
  - Flexibility and ease of deployment are bad news
  - Code + configuration files, policies, databases

- Example: low-level IP security is secure, reliable and efficient but...
  - Applications are often unaware of security mechanisms
  - Security relies on IP addresses, hard to interpret in terms of applications
  - IPSEC policies are delicate
  - Implementations are only starting to interop
Web Services
What’s a Web Service?

“A web service is a web site intended for use by computer programs instead of human beings.” (Barclay et al)

So XML not HTML

Service messages in SOAP format:
- Envelope/Header – addressing, security, and transactional headers
- Envelope/Body – actual payload

Winer proposed SOAP in 1998; by 2006, very widely deployed
- On Internet: web search (eg Google), storage (eg Amazon S3), etc
- On intranets, interconnect between business systems
A sample service

My Laptop

```
Service s = new BookshopService();
foreach (int orderId in PendingOrders)
{
  ...;
  s.GetOrder(orderId).display();
}
```

Implementation via proxy class and HTTP transport

The Internet HTTP/TCP/IP

```
[WebMethod]
Order GetOrder(int orderId)
{
  ...
  SQL.select(orderId)
  ...
}
```

SOAP Request

SOAP Response

My Bookshop

Implementation via WebService classes in Web Server

Order database
A sample service (XML wire format)

Implementation via proxy class and HTTP transport

SOAP Request:

```xml
<Envelope>
  <Header>
    <Action>http://bookshop/ws/OrderStatus</Action>
    <To>http://www.bookshop.com/ws/orderstatus.asmx</To>
    <MessageId>uuid:5ba86b04-3c0f-17804286fe40</MessageId>
  </Header>
  <Body>
    <GetOrder>
      <orderId>20</orderId>
    </GetOrder>
  </Body>
</Envelope>
```

Response

```xml
[WebMethod]
Order GetOrder(int orderId)
{
    ...;
    SQL.select(orderId);
    ...
}
```
Origins and features

- Prior RPC mechanisms (e.g., CORBA, DCOM) had interop problems, and were often blocked by firewalls.
- SOAP originates in the idea of “RPC using XML over HTTP.”
  - Achieve interop and reach via success of XML and HTTP.
  - Championed by IBM/MS since around 2000.
- Features (a mixed bag, from a security perspective):
  - **Standards-based** – XML, HTTP, XML-DSIG, XML-ENC.
  - **Composable specifications** – optional features such as addressing, security, reliability assembled per service.
  - **Vendor interoperability** – “interop fests” from the start.
  - **Message-centric** – based on asynchronous messaging, intended to be tolerant of latencies and failures.
  - **App-level (SOAP) security** as well as transport (SSL).
Web Services security

- “SOAP level security aims to provide end-to-end, compositional application-level security, independently of transport protocol”

- A grammar for SOAP-based security protocols
  - Automated processing of security headers
  - Informal semantics except for XML syntax
  - Security tokens = wire format for claims and evidence
    - Keys, certificates, x509 signatures, Kerberos tickets,…

- Fresh standards
  - WS-Trust, WS-SecureConversation, WS-SecurityPolicy,…

- Fresh implementations
  - At Microsoft: WSE → Windows Communications Foundation
The WS-Security header

- Supports application-level security
  - Partially encrypted, signed messages less likely to be blocked at firewall than encrypted SSL tunnel
  - But much less efficient than SSL
- Timestamp – to help prevent replay attacks
- Tokens identifying principals and keys
  - Username token: name and password
  - X509: name and public-key
  - Others including Kerberos tickets, and session keys
- Signatures – XML-DSIG standard
  - Bind together list of message parts
  - May be logged and produced as evidence, unlike SSL traffic
- Encrypted Keys – XML-ENC standard
  - May be used to encrypt other parts of the message
(1) Password-based signature

UsernameToken assumes both parties know adg’s secret password $p$

Each DigestValue is the sha1 hash of the URI target

$hmacsha1(key, SignedInfo)$ where $key \approx psha1(p + nonce + created)$

<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=1>
        <Username>"adg"
        <Nonce>"mTbzQM84RkFqza+lIes/xw=="
        <Created>"2004-09-01T13:31:50Z"
      <Signature>
        <SignedInfo>
          <SignatureMethod Algorithm=hmac-sha1>
          <Reference URI=#2>
            <DigestValue>"U9sBHidIkVvKA4vZo0gGKxMhA1g="
            <SignatureValue>"8/ohMBZ5JwzYyu+POU/v879R01s="
          <KeyInfo>
            <SecurityTokenReference>
              <Reference URI=#1 ValueType=UsernameToken>
                  <Body Id=2>
                    <StockQuoteRequest>
                      <symbols>
                        <Symbol>"FABRIKAM"
                        <Symbol>"CONTOSO"
Actually, to prevent various attacks, it’s necessary to co-sign other message parts with the body.
The X.509 standard is a remnant of the now abandoned early 80s Open Systems Interconnection (OSI) project, but is now widely used with Internet standards such as SSL/TLS.

An X.509 certificate binds a public key to a human-readable subject name, and is signed by an issuer.

An X.509 cert may include the following fields:
- issuer’s and subject's name (an X.500 directory name)
- validity period (start and end date in UTC)
- subject's public key (algorithm and public key data)
- issuer’s and subject’s generic name (eg domain, URI)
- identifier for issuer’s policy (how was the identity verified?)
- issuer’s signature for the entire certificate
(2) X.509-based signature

X.509 is an ASN.1 format not XML, so needs to be binary encoded

rsasha1(key, SignedInfo) where key is private key corresponding to the public key in X.509 cert
(3) X.509-based encryption

<Envelope>
  <Header>
  <Security>
    <EncryptedKey>
      <EncryptionMethod Algorithm=rsa-1_5>
      <KeyInfo>
        <SecurityTokenReference>
          <KeyIdentifier ValueType=X509SubjectKeyIdentifier>="bBwPfItvKp3b6TNDq+14qs58VJQ="
        </KeyIdentifier>
        <CipherData>
          <CipherValue>
            "gXWRbUNSo7H5EeAO9GhE7nrq5VdTjScMFbiftmW...
          </CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI=#2>
        </ReferenceList>
      </KeyInfo>
    </EncryptedKey>
  </Security>
  <Body>
    <EncryptedData Id=2 Type=Content>
      <EncryptionMethod Algorithm=aes128-cbc>
        <CipherData>
          <CipherValue>
            "v8XMS3XmttksWJDTnCJ86lxPW1L0cA+s16nFQgNM..."
          </CipherValue>
        </CipherData>
      </EncryptionMethod>
    </EncryptedData>
  </Body>
</Envelope>

rsa-enc (key, K) where key is the server’s public key and K is a fresh key

aes-enc (body, K)
Attacks on Web Services security
Attacks on SOAP security

- Web services vulnerable to same sorts of attacks as websites
  - Buffer overruns, denial of service, SQL injection, etc
- New concerns: flexible, XML-based protocols
  - Web services developers can design and deploy their own application-specific security protocols
  - XML message format open to rewriting attacks
    - Much like classic active attackers (Needham-Schroeder’78)
    - Attacker can redirect, replay, modify, impersonate
    - New: message processing is driven by a flexible, semi-structured message format
- This flexibility is bad news for security
  - We found a range of problems in specs & code, thus motivating research on theory and tools
A Signed SOAP message before...

<Message to bank’s web service says: “Transfer $1000 to Bob, signed Alice”>

Bank can verify the signature has been computed using key derived from Alice’s secret password

Lightly edited XML as sent by WSE 1.0

```xml
<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=2>
        <Username>Alice</Username>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</Nonce>
        <Created>2003-02-04T16:49:45Z</Created>
      </UsernameToken>
      <Signature>
        <SignedInfo>
          <Reference URI= '#1'><DigestValue>Ego0...</DigestValue></Reference>
          <SignatureValue>vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</SignatureValue>
          <KeyInfo>
            <SecurityTokenReference>
              <Reference URI=#2/>
            </SecurityTokenReference>
          </KeyInfo>
        </SignedInfo>
      </Signature>
    </Security>
  </Header>
  <Body Id=1>
    <TransferFunds>
      <beneficiary>Bob</beneficiary>
      <amount>1000</amount>
    </TransferFunds>
  </Body>
</Envelope>
```
... and after an XML rewriting attack

Charlie has intercepted and rewritten this message

```
<Envelope>
  <Header>
    <Security>
      <UsernameToken Id=2>
        <Username>Alice</>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
        <Created>2003-02-04T16:49:45Z</>
      </UsernameToken>
    </Security>
    <Signature>
      <SignedInfo>
        <Reference URI= #1><DigestValue>Ego0...</> </Reference>
        <SignatureValue>vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
      </SignedInfo>
      <KeyInfo>
        <SecurityTokenReference><Reference URI=#2/>
        <BogusHeader>
          <Body Id=1>
            <TransferFunds>
              <beneficiary>Bob</>
              <amount>1000</>
            </TransferFunds>
          </Body>
          <Body>
            <TransferFunds>
              <beneficiary>Charlie</>
              <amount>5000</>
            </TransferFunds>
          </Body>
        </BogusHeader>
      </KeyInfo>
    </Signature>
  </Header>
</Envelope>
```

The indirect signature of the body, now hidden in BogusHeader, may still appear valid.

Although Alice’s password has not been broken, the message now reads “Transfer $5000 to Charlie, signed Alice.”
“Every message should say what it means”

- To prevent deceitful use of messages, make explicit (and check) any information used to process the message.
- In contrast, what the envelope says...
  - is quite complicated
  - is more general than its intended meaning (by design)
  - provides many details on mechanisms from various specs
  - may not contain all relevant, authenticated information

- It is very easy to design & deploy new protocols for web services.
- It is getting much harder to verify those protocols.
Another XML rewriting attack

Alter and replay envelopes to confuse participants

From: Alice
To: Bookshop
(signed by Alice)

Sent: Monday
From: Alice
To: Bank
Action: “Pay Charlie $20”
(signed by Alice)

Sent: Tuesday
From: Alice
To: Bank
(signed by Alice)

Sent: Wednesday
From: Alice
To: Bookshop
(signed by Alice)

Alice’s laptop

Alice’s bookshop
(Web Service)

Someone on the net
(Charlie?)
An overzealous service

Someone on the net (Charlie?)
Suppose A sends an important proof, encrypted for a journal editor J, and then signed; even though M cannot decrypt the proof, M can steal the credit for the proof by replacing A’s signature with their own.

**Principle 5** When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. ...

Abadi and Needham *Prudent engineering practice for cryptographic protocols* 1994
A Leak with Sign-Then-Encrypt

- XML-DSIG stipulates that the hashes of each message part be sent along with the signature
- If the body is signed, then encrypted, the hash of the body is sent in the clear
- If the body has low entropy, an attacker can mount an offline guessing attack
- One fix is to omit the redundant hashes, but this violates the DSIG standard
- Another fix is to encrypt the whole signature - that’s right, Sign Before Encrypt And Encrypt Signature!
Some Tools for Web Services Security

The Samoa project, 2003— http://securing.ws
We designed TulaFale, a programming language to model WSE protocols and hand-wrote models for a series of WS specs and WSE protocols.

TulaFale = pi + XML + predicates + assertions

What TulaFale does

- TulaFale script
- Predicate library
- Intermediate pi-calculus
- ProVerif Analyzer [B. Blanchet]

OK, or No because...
TulaFale Demo: A Secure RPC
A secure RPC

- A typical system model:
  - A single certification authority (CA) issuing X.509 public-key certificates for services, signed with the CA's private key.
  - Two servers, each equipped with a public key certified by the CA and exporting an arbitrary number of web services
  - Multiple clients, acting on behalf of human users

- Threat model: an active attacker, in control of network, but knowing none of:
  - The private key of the CA
  - The private key of any public key certified by the CA
  - The password of any user in the database

- Security goals: authentication of each message; and correlation of request and response; but not confidentiality
An intended run of the protocol

Msg 1 includes signature of S,id1,t1,b1 under key derived from username token for U

Msg 2 includes signature of id1,id2,t2,b2 under public key of S
**pi+XML+predicates+assertions**

**Predicate** `envl` defined by Horn clauses with message patterns:

```
msg1 =
  <Envelope>
    <Header>
      <To>uri</To>
      <Action>ac</Action>
      <MessageId>id1</MessageId>
      <Security>
        <Timestamp><Created>t1</Created></Timestamp>
        eutok
        sig1
      </Security>
    </Header>
  </Envelope>.
```

For example, this predicate is used in two ways, to construct and parse Message 1.

TulaFale messages are terms in a many-sorted algebra with sorts:

- item, items, att, atts, bytes, string
predicate isMsg1(msg1:item,U:item,sx:bytes,cert:bytes,S:item, id1:string,t1:string,b1:item) :-

env1(msg1,uri,ac,id1,t1,eutok,sig1,b1),
S = <Service><To>uri</To><Action>ac</Action><Subject>subj</Subject></Service>,
isEncryptedData(eutok,utok,sx),
isUserTokenKey(utok,un,t1,sk),
isSignature(sig1,"hmacsha1",sk,
<list>
  <Body>b1</Body>
  <To>uri</To>
  <Action>ac</Action>
  <MessageId>id1</MessageId>
  <Created>t1</Created>
eutok</list>).

TulaFale library includes predefined predicates for XML signatures and encryption

For example, this predicate uses these predicates to check structure of Message 1
pi + XML + predicates + assertions

By sending a message on init, the attacker can pick any payload and destination.

Each **begin-event** marks the intent to send a message.

Messages are exchanged on a public SOAP channel.

Each **end-event** marks the intent to accept a message as valid.

```plaintext
channel init(item, bytes, bytes, string, item).

process Client(k: bytes, U: item) =
  in init (S, certA, n, t1, b1);
  new id1: string;
  begin C1(U, <list>S id1 t1 b1</>);
  filter mkMsg1(msg1, U, S, k, certA, n, id1, t1, b1) → msg1;

  out soap(msg1);
  in soap(msg2);
  filter isMsg2(msg2, S, k, id1, id2, t2, b2) → id2, t2, b2;
  end C2(U, <list>U S id1 t1 b1 id2 t2 b2</>);
  done.
```
new \texttt{sr:bytes}; \textbf{let} \texttt{kr = pk(sr)};
new \texttt{sx1:bytes}; \textbf{let} \texttt{cert1 = x509(sr, "BobsPetShop", "rsasha1", pk(sx1))};
new \texttt{sx2:bytes}; \textbf{let} \texttt{cert2 = x509(sr, "ChasMarket", "rsasha1", pk(sx2))};
\textbf{out} \texttt{publish(base64(kr))};
\textbf{out} \texttt{publish(base64(cert1))};
\textbf{out} \texttt{publish(base64(cert2))};
\textbf{( !MkUser(kr) | !MkService(sx1,cert1) | !MkService(sx2,cert2) |}
\textbf{ (!in anyUser(U); Client(kr,U)) |}
\textbf{ (!in anyService(sx,cert,S); Server(sx,cert,S)) )}

The implicit attacker, running in parallel, can:
- Send and receive on the soap channel
- Generate arbitrarily many users and services
- Initiate arbitrarily many sessions
Some Tulafale queries

query $\text{end}:C2(U,m12)$.  
query $\text{end}:C1(U,m1)$.  
query $\text{end}:C2(U,m12) \Rightarrow \text{begin}:C2(U,m12)$.  
query $\text{end}:C1(U,m1) \Rightarrow \text{begin}:C1(U,m1)$.

We also run basic reachability queries (sanity checks)

We verify two correspondence properties from $\text{end-events}$ to $\text{begin-event}$ with matching contents (including both messages for C2)
Suppose a client does not sign the message identifier $id1$...

Pair $(id1, t1)$ uniquely identifies the message only if $id1$ and $t1$ are signed.

We found and fixed faults like this in preliminary WSE samples.
If the client doesn’t generate fresh id1’s, then message correlation (C2) fails; the tool easily finds this bug.
What about insider attacks?

If one or more passwords are compromised, there is an insider attack on message correlation; more extensive changes to the script are needed to model this.
Vérification des Protocoles Cryptographiques et de leurs Implémentations

1. Modélisation des protocoles en pi calcul
2. Comment vérifier l’usage des protocoles? applications aux services Web (outils, demo)
3. Comment vérifier leurs implémentations? (demo)
4. Cryptographie formelle/concrete
We designed TulaFale, a programming language to model WSE protocols and hand-wrote models for a series of WS specs and WSE protocols.

\[ \text{TulaFale} = \pi + \text{XML} + \text{predicates} + \text{assertions} \]
Secure Conversations
Secure conversations

- WS-Security provides basic mechanisms to secure SOAP traffic, one message at a time
  - Signing and encryption keys derived from long-lived secrets like passwords or private keys

- If a SOAP interaction consists of multiple, related messages, WS-Security alone may be inefficient, and does not secure session integrity
  - Standard idea: establish short-lived session key

- Recent specs describe this idea at the SOAP-level
  - **WS-SecureConversation** defines *security contexts*, used to secure sessions between two parties
  - **WS-Trust** defines how security contexts are issued and obtained
A typical system

STS = Security Token Server
RST = Request Security Token
RSTR = RST Response
SC = Security Context
SCT = SC Token
Open-ended conversations

- We prove authentication for whole sessions
- We rely on some combination of manual and automated proofs

for \( n \neq 0 \)
Discussion

- A formal analysis of WS-Trust and WS-SecureConversation
  - XML syntax and automation very effective, against a demanding, realistic attacker model
  - Approx 1000 lines of script – too large for manual proofs

- As is common, these specs:
  - focus on message formats for interoperability
  - are non-committal regarding security, for example, no clear spec of contents of SCs

- By making modes, data, and goals explicit, we found design and implementation bugs
Policy-Based Security
Policy-based security

- Clients, services use XML files to pick security mechanisms

```xml
<Policy Id="Msg1">
  <All>
    <Confidentiality>
      <TokenInfo>
        <SecurityToken>
          <TokenType>X509v3</TokenType>
          <Claims><SubjectName>S</SubjectName></Claims>
        </SecurityToken>
        <MessageParts>Body()</MessageParts>
      </TokenInfo>
    </Confidentiality>
    <Integrity>
      <TokenInfo>
        <SecurityToken>
          <TokenType>UsernameToken</TokenType>
          <Claims><SubjectName>U</SubjectName></Claims>
        </SecurityToken>
        <MessageParts>Body() Header("To") Header("MessageId")</MessageParts>
      </TokenInfo>
    </Integrity>
  </All>
</Policy>
```
Gathering policy configurations

Automated tools for collecting, parsing policies from IIS Servers, Clients
Config = [Policy1, Policy2, Policy3, Policy4]
Tool 2: Policy Generator/Analyzer

In WSE 2.0, WS-SecurityPolicy files drive security; hence, we can generate TulaFale directly from implementation files.

What our tools do

Generator $C(-)$

Analyzer $S(-,-)$

Static warnings

WSE 2.0 out of the box

code $C\#$/VB

policy config $C(L)$

WSE 2.0

CLR (IL)

SOAP processing

spec $L$ of a secure link

Generator $C(-)$

Analyzer $S(-,-)$

Static warnings

TulaFale script $S(C(L),L)$

predicate library

TulaFale

ProVerif (pi calculus)

OK, or No because...
Translating policies to predicates

<Policy Id="Msg1">
  <All>
    <Confidentiality>
      <TokenInfo>
        <SecurityToken>
          <TokenType>X509v3</TokenType>
          <Claims><SubjectName>S</SubjectName></Claims>
        </SecurityToken>
      </TokenInfo>
      <MessageParts>Body()</MessageParts>
    </Confidentiality>
    <Integrity>
      <TokenInfo>
        <SecurityToken>
          <TokenType>UsernameToken</TokenType>
          <Claims><SubjectName>U</SubjectName></Claims>
        </SecurityToken>
        <MessageParts>Body() Header("To")
                      Header("MessageId")</MessageParts>
      </SecurityToken>
    </Integrity>
  </All>
</Policy>

Conjunction

Encryption Requirement

Signature Requirement

predicate hasMsg1Policy(msg1:item,U:item,pwd:string,
                        S:item,skS:bytes,id1:string,req:item) :-
msg1 = <Envelope>
  <Header>
    <To>S</To>
    <MessageId>id1</MessageId>
  </Header>
  <Security>utok
    sig1</Security>
  <Body>b1</Body>,
isEncryptedData(b1,req,skS),
isUserTokenKey(utok,U,pwd,skU),
isSignature(sig1,"hmacsha1",skU, [<Body>b1</Body> <To>S</To> MessageId:id1])}.
Security for all generated configs?

- **Theorem:** All policy configurations generated from link specs enforce their security goals
  - For all link specifications L, S(C(L),L) provides request & response authentication, correlation, and secrecy.
  - Hence, at least generated configs can be safely deployed

- **Proof:**
  - Non-obvious: there are infinitely-many link specs
  - Use combination of automated proofs and manual reasoning
  - Hint 1: Reduce to 4 base kinds of link-generated policies
  - Hint 2: Prove that configs with all links enabled is secure (programmed unfoldings in the pi calculus)
Security for any client policy?

- Theorem: If a service uses a link-generated policy, then irrespective of the client policies, the resulting configuration preserves request authentication and response secrecy.

- Hence, naïve clients cannot break service authentication.

- Proof:
  - Combination of automated proofs and manual reasoning.
  - Hint: Even the weakest send policy preserves secrecy of passwords and signing keys.
Tool 3: WSE/WCF Policy Advisors

Advisor guesses intended goals and runs queries that check for:

1. likely errors in configuration file settings
2. conformance to conservative policy schema
3. likely errors in (request, response, fault) mappings
4. likely errors in particular policies

**WSE 2.0 SP2 out of the box**

- code C#/VB
- policy config
- WSE 2.0
  - CLR (IL)
  - SOAP processing

**Our plug-in**

- static queries
- security report
- WSE Policy Advisor
One-line summary of each triggered query

Details of the risk and the remedial action

Browser shows relevant part of policy file

Released Feb 2005; WSE Config Editor can invoke advisor since WSE2 SP2
Recommended by MS Patterns and Guidance for web services; SWS’05 paper
Policy Advisor for WSE 3.0

- XSL Stylesheet, ships with WSE 3.0, Nov ’05

**Advisories**

Test root certificates are allowed.

- StockService [config]

**Risk:** Any usage of X.509 certificates for signing or encrypting is unsafe. An active attacker can generate valid test certificates, then for instance use these certificates to sign any message.

**Advice:** Do not use test keys in production: set the attribute allowTestRoot="false" in the <x509> element of the WSE configuration file.

---

This policy enables a dictionary attack on an encrypted request, response, or fault whose message body contains the phrase "system error".

- StockService (policy: MySecurityPolicy) (SOAP: request) [policyCache]
- StockService (policy: MySecurityPolicy) (SOAP: response) [policyCache]
- StockService (policy: MySecurityPolicy) (SOAP: fault) [policyCache]

**Risk:** The message body is encrypted, but the cryptographic hash of the plaintext message body is also included in the signature. Hence, an attacker that intercepts the message may obtain this hash and compare it to the hash of a large number of potential message bodies. Once two hashes match, the attacker has broken confidentiality of the message body.

**Advice:** If the body cannot be guaranteed to have high entropy (that is, if the body does not always include some fresh, secret cryptographic value), use either messageProtectionOrder="EncryptBeforeSign" or messageProtectionOrder="SignBeforeEncryptAndEncryptSignature".
Summary on application security

- Web services security enables extreme flexibility
  - Specs and implementations are just emerging
  - Attacks and proofs are subtle: tool support needed

- We bridge the gap between theoretical threat model and XML as used in WS security protocols
  - Put effort into real samples & implementations, found bugs
  - Obtained theorems about wire-level protocols

- We develop tools for the automated analysis of security for systems based on crypto protocols
  - Proving protocols secure in isolation is not enough
  - Our tools find attacks, verify configs, generate safe configs

- Good place to develop formal tools, get positive results
  - Standard message formats, composition, wide applicability
Verifying Protocol Implementations

Where is the model coming from?
What about the implementation?
Models vs implementations

- Protocol specifications remain largely informal
  - They focus on message formats, not on local enforcement of security properties

- Formal models are short, abstract, hand-written
  - They ignore large functional parts of implementations
  - Their formulation is driven by verification techniques
  - It is easy to write models that are safe but dysfunctional (testing & debugging is difficult)

- Specs, models, and implementations drift apart...
  - Even informal synchronization involves painful code reviews
  - How to keep track of implementation changes?
From code to model

- Our approach: we automatically extract models from protocol implementations
  - We consider reference implementations, not (yet) production code

- Executable code is more detailed than models
  - Some functional aspects can be ignored for security
  - Model extraction can safely erase those aspects

- Executable code has better tool support
  - Types, compilers, debuggers, libraries, other verification tools
One source, three tasks

- **Application**
  - My code
  - Authz
  - Other Libraries

**My protocol**

**Source code with modules and strong interfaces**

- **Symbolic Crypto**
  - Symbolic verification
  - Symbolic testing & debugging

- **Concrete Crypto**
  - Platform (CLR)
  - Interoperability (via SOAP)

- **Other Libraries**
  - Some other implementation (WSE)
  - ProVerif
  - fs2pv

- **Source code with modules and strong interfaces**

**Libraries**

- Platform (CLR)
- Crypto Net

**API**

- Interoperability (via SOAP)
1. Symbolic testing and debugging

- My code
- Authz
- Other Libraries

Coded in C#, F#...

- My protocol
- Application

We use idealized cryptographic primitives

Safety relies on typing

Symbolic Crypto

Attacker (test)

Platform (CLR) Crypto Net

We model attackers as arbitrary code with access to selected libraries
2. Formal verification

We only support a subset of F#.

We model attackers as arbitrary code with access to selected libraries.

Formal verification considers ALL such attackers.

Pass: Ok for all attackers, or No + potential attack trace.

Translated to pi calculus.

Secrets and crypto libraries:

- fs2pv
- ProVerif
- Symbolic Crypto

Application components:

- My code
- Authz
- Other Libraries

Attacker (unknown)
3. Concrete testing & interop

Coded in C#, F#...

My code
My protocol
Application
Authz
Other Libraries
Concrete Crypto
Crypto Net
Platform (CLR)

We test that our code produces and consumes the same messages as another implementation.

We only change our implementation of cryptographic primitives.

Attacker (test)
We can still run attacks to test other implementations.

Some Other Implementation (WSE)
Interoperability (via SOAP)
Source language: F#

- F#, a variant of ML for the CLR, by Don Syme
  
  [http://research.microsoft.com/fsharp](http://research.microsoft.com/fsharp)
  “Combining the strong typing, scripting and productivity of ML with the efficiency, stability, libraries, cross-language working and tools of .NET.”

- Experimental language for research and prototyping

- Clean strongly-typed semantics
  - We rely on abstract interfaces
  - We use algebraic data types and pattern-matching for symbolic cryptography, for XML applications
Target verification tool: ProVerif

- ProVerif, an automated cryptographic protocol verifier developed by Bruno Blanchet (ENS)
- Input: protocol scripts written in applied pi calculus
  - Concurrent processes + parametric cryptography
- What it can prove:
  - Secrecy, authenticity (correspondence properties)
  - Equivalences (e.g. protection of weak secrets)
- How it works (flavor):
  - Internal representation based on Horn clauses
  - Resolution-based algorithm, with clever selection rules
  - Attack reconstruction


Password-based MAC (Demo)

Coding and verifying a one-message protocol
Password-based authentication

\[ A \rightarrow B : \quad \text{HMACSHA1}(nonce, pwd_A |text), \]
\[ \text{RSAEncrypt}(pk_B, nonce), \]
\[ text \]

- A simple, one-message authentication protocol (simplified from a WS-Security sample protocol)

- Two roles
  - client (A) sends some text, along with a MAC
  - server (B) checks authenticity of the text
  - the MAC is keyed using a nonce and a shared password
  - the password is protected from dictionary attacks by encrypting the nonce with the server’s public key
Making and verifying messages

\[ A \rightarrow B : \quad \text{HMACSHA1}(nonce, pwd_A | text), \]
\[ \quad \text{RSAEncrypt}(pk_B, nonce), \]
\[ \quad text \]

```
let mac n pwd text = hmacsha1 n (concat (utf8 pwd) (utf8 text))

let make text pke pwd =
  let nonce = mkNonce() in
  (mac nonce pwd text, rsa_encrypt pke nonce, text)

let verify (m,en,text) skd pwd =
  let nonce = rsa_decrypt skd en in
  if not (m = mac nonce pwd text) then failwith "bad MAC"
```
let address = S "http://server.com/pwdmac"
let pwdA = Prins.getPassword(S "A")
let pkB = Prins.getPublicKey(S "B")

let client text =
  log(Send(text));
  Net.send address (marshall (make text pkB pwdA))

let skB = Prins.getPrivateKey("B")
let server () =
  let m,en,text = unmarshall (Net.accept address) in
  verify (m,en,text) skB pwdA; log(Accept(text))
One source, three tasks

- Using concrete libraries, our client and server run using TCP
  
  Sending FADCIZhW3XmgUABgRJj1KjnWyDvEoAAe...

- Using symbolic libraries, we can see through cryptography
  
  Sending HMACSHA1{nonce3}['pwd1' | 'Hi']
  | RSAEncrypt{PK(rsa_secret2)}[nonce3] | 'Hi'

- Using symbolic libraries, fs2pv generates a ProVerif model
  
  RESULT Accept(x) ==> Send(x) is true.
Two implementations of Crypto

```fsharp
module Crypto // concrete code in F#
open System.Security.Cryptography

type bytes = byte[]

let rng = new RNGCryptoServiceProvider ()
let mkNonce () =
    let x = ByteArray.make 16 in
    rng.GetBytes x; x

let hmacsha1 k x =
    new HMACSHA1(k).ComputeHash x

let rsa = new RSACryptoServiceProvider()
let rsa_keygen () = ...
let rsa_pub (RSA r) = ...
let rsa_encrypt (RSA r) (v:bytes) = ...
let rsa_decrypt (RSA r) (v:bytes) =
    rsa.ImportParameters(r);
    rsa.Decrypt(v,false)

module Crypto // symbolic code in F

type bytes =
| Name of Pi.name
| HmacSha1 of bytes * bytes
| RsaKey of rsa_key
| RsaEncrypt of rsa_key * bytes

let freshbytes label = Name (Pi.name label)
let mkNonce () = freshbytes "nonce"

let hmacsha1 k x = HmacSha1(k,x)

let rsa_keygen () = SK (freshbytes "rsa")
let rsa_pub (SK(s)) = PK(s)
let rsa_encrypt s t = RsaEncrypt(s,t)
let rsa_decrypt (SK(s)) e = match e with
    | RsaEncrypt(pke,t) when pke = PK(s) → t
    | _ → failwith "rsa_decrypt failed"
```
Formalizing a subset of F#

How to justify using ProVerif for proving F# properties?
A first-order functional language

\[ M, N ::= \]
\[ x \] \hspace{1cm} value
\[ a \] \hspace{1cm} variable
\[ f(M_1, \ldots, M_n) \] \hspace{1cm} name
\[ e ::= \] \hspace{1cm} constructor application
\[ M \] \hspace{1cm} expression
\[ ℓ M_1 \ldots M_n \] \hspace{1cm} value
\[ \text{fork}(\text{fun}() \rightarrow e) \] \hspace{1cm} function application
\[ \text{match } M \text{ with } (M_i \rightarrow e_i)_{i \in 1..n} \] \hspace{1cm} fork a parallel thread
\[ \text{let } x = e_1 \text{ in } e_2 \] \hspace{1cm} match: \( M_i \) patterns, \( n \geq 0 \)
\[ d ::= \] \hspace{1cm} sequential evaluation
\[ \text{type } s = (f_i \text{ of } s_{i1} \ldots s_{im_i})_{i \in 1..n} \] \hspace{1cm} declaration
\[ \text{let } x = e \] \hspace{1cm} datatype declaration
\[ \text{let } ℓ x_1 \ldots x_n = e \hspace{1cm} n > 0 \] \hspace{1cm} value declaration
\[ S ::= d_1 \ldots d_n \] \hspace{1cm} function declaration
\[ \text{system: list of declarations} \]
Expressing security goals

Authentication queries are of the form \( \text{ev}: E \Rightarrow \text{ev}: B_1 \lor \cdots \lor \text{ev}: B_n \).

\( C \models \text{query ev}: E \Rightarrow \text{ev}: B_1 \lor \cdots \lor \text{ev}: B_n \) if and only if whenever \( C \equiv \text{event } E \sigma \mid C' \), there is \( C'' \) and \( i \in 1..n \) such that \( C' \equiv \text{event } B_i \sigma \mid C'' \).

The system \( S \) is safe for \( q \) if and only if, whenever \( S \rightarrow^{*} C \), we have \( C \models q \).

We write \( I \vdash S : I' \) to mean that system \( S \) assumes an implementation of interface \( I \), and exports the interface \( I' \).

We write \( S :: I_{\text{pub}} \) to mean that \( \text{Prim} \vdash S : I_{\text{pub}}, I_{\text{priv}} \) for some \( I_{\text{priv}} \).

An opponent \( O \) for \( S :: I_{\text{pub}} \) is any system with \( \text{Prim} \log, I_{\text{pub}} \vdash O \).

\( S :: I_{\text{pub}} \) is robustly safe for \( q \) when \( S :: I_{\text{pub}} \) and \( S O \) is safe for \( q \) for all opponents \( O \).
Authentication (for the example)

Let $S$ be the system consisting of application code and symbolic libraries. Let $I_{pub}$ be the interface

Net.send: fun 2, Net.accept: fun 1,
Crypto.S: fun 1, Crypto.iS: fun 1,
Crypto.base64: fun 1, Crypto.ibase64: fun 1,
Crypto.utf8: fun 1, Crypto.iutf8: fun 1,
Crypto.concat: fun 1, Crypto.iconcat: fun 1,
Crypto.concat3: fun 1, Crypto.iconcat3: fun 1,
Crypto.mkNonce: fun 1, Crypto.mkPassword: fun 1,
Crypto.rsa_keygen: fun 1, Crypto.rsa_pub: fun 1,
Crypto.rsa_encrypt: fun 2, Crypto.rsa_decrypt: fun 2,
Crypto.hmacsha1: fun 2,
pkB: val, client: fun 1, server: fun 1

We verify that $S :: I_{pub}$ is robustly safe for $ev:Accept(x) \Rightarrow ev:Send(x)$
Mapping F to a Verifiable Model
How to compile a function?

- Our compiler specifically targets symbolic verification
- We select a translation for each function
  - Complete inlining (anticipating resolution)
  - ProVerif reductions (eliminated by ProVerif)
  - ProVerif predicate declarations (logic programming)
  - ProVerif processes
    - We follow Milner’s classic “functions as processes”
    - Each call takes two channel-based communication steps
    - We use private or public channels depending on the interface
How to compile a function?

Consider the F# function

```fsharp
let mac nonce pwd text =
    Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

We can translate it as a process

```fsharp
!in(mac, (nonce,pwd,text,k));
    out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text)))))
```

We actually translate `mac` into a ProVerif reduction rule:

```fsharp
reduc mac(nonce,pwd,text) =
    HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text)))
```
Experimental Results
Experimental results

- We coded and verified a series of protocols and libraries
  - An implementation of Otway-Rees
  - Libraries for principals + realistic attacker models
  - Libraries for Web Services Security standards
  - A series of Web Services sample protocols

- We tested interoperability with other implementations of web services protocols (WSE, WCF)
  - We can use our command-line client
    + client application code in C#
    + an IIS/WSE web server
  - We can register an IIS/F# SOAP filter for our server
    + client application code in C# using WSE
## Experimental results

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOCs</td>
</tr>
<tr>
<td>password-based MAC</td>
<td>38</td>
</tr>
<tr>
<td>password-based MAC variant</td>
<td>75</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>148</td>
</tr>
<tr>
<td>WS password signing</td>
<td>85</td>
</tr>
<tr>
<td>WS X.509 signing</td>
<td>85</td>
</tr>
<tr>
<td>WS password-based MAC</td>
<td>85</td>
</tr>
<tr>
<td>WS request-response</td>
<td>149</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Security Goals</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>queries</td>
<td>secrecy</td>
</tr>
<tr>
<td>password-based MAC</td>
<td>4</td>
<td>weak pwd</td>
</tr>
<tr>
<td>password-based MAC variant</td>
<td>5</td>
<td>pwd</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>16</td>
<td>key</td>
</tr>
<tr>
<td>WS password signing</td>
<td>5</td>
<td>no</td>
</tr>
<tr>
<td>WS X.509 signing</td>
<td>5</td>
<td>no</td>
</tr>
<tr>
<td>WS password-based MAC</td>
<td>3</td>
<td>weak pwd</td>
</tr>
<tr>
<td>WS request-response</td>
<td>15</td>
<td>no</td>
</tr>
</tbody>
</table>
Conclusions
Limits of our approach

- As usual, formal security guarantees hold only within the boundaries of their model
  - We keep model and implementation in sync
  - We automatically deal with very precise models
  - We can precisely “program” the attacker model

- We verify our own implementations, not legacy code

- We trust the F# compiler and the .NET runtime
  - Certification is possible, but a separate problem

- We trust our symbolic model of cryptography
  - This is the common Dolev-Yao assumption
  - Partial computational soundness results may apply
Related work

- Going in the opposite direction, several tools generate executable code from formal models of protocols:
  - CAPSL: Muller and Millen (2001)
  - The resulting code seldom works with other implementations

- In other work, we compile models from declarative configuration files (WS-SecurityPolicies)

- Giambagi and Dam show conformance between models and their implementation, in terms of information flows

- Goubault-Larrecq and Parennes (2005) derive models from cryptographic code (in C)
Summary on verified implementations

- We verify reference implementations of security protocols.
- Our implementations can run with both concrete and symbolic cryptographic libraries.
  - The concrete implementation is for production and interop testing.
  - The symbolic implementation is for debugging and verification.
- We develop our approach for protocols written in F#, running on the CLR, verified by ProVerif.
  - We show its correctness for a range of security properties, against realistic classes of adversaries.
  - We apply our approach to protocols for web services security.