BPEL Orchestration of Secure WebMail

Saket Kaushik
ISE Department
George Mason University
Fairfax, VA 22030
+1-703-993-1632
skaushik@gmu.edu

Duminda Wijesekera
ISE Department
George Mason University
Fairfax, VA 22030
+1-703-993-1578
dwijesek@gmu.edu

Paul Ammann
ISE Department
George Mason University
Fairfax, VA 22030
+1-703-993-1660
pammann@gmu.edu

ABSTRACT

WebMail proposes to migrate existing SMTP-based mail systems to Web-Services. We show how a verifiably-correct, generic mail service that enables extensions of SMTP-based standard mail use cases that avoids known misuse cases can be specified using WSDL and orchestrated using BPEL.

Categories and Subject Descriptors

C.2.0 [**Computer and Communication Networks**]: General – *Security and Protection*; C.2.2 [**Network Protocols**]: *Applications;* D.2.4 [**Software/Program Verification**]: *Applications*

General Terms

Design, Security, Verification.

Keywords

WSEmail, WebMail, BPEL, SMTP use cases, SMTP misuse cases, verification.

1. INTRODUCTION

The utility, security and trustworthiness of conventional email system are being questioned on account of its increasing misuse by 'spammers' and fraudsters. Based on suggestions to replace existing SMTP-based [12,13] system with a secure system – such as WSEmail [17] – we propose using *web services definition language* (WSDL) [7] to specify a web-based customizable emailsystem that gives recipients control over email delivery. Consequently, the proposed system enables more Use Cases than the conventional SMTP-based mail. Given that conventional email and our extensions can be used with mal-intent, we specify a collection of Misuse Cases that are shown to be thwarted by our design. Example standard Use Cases and Misuse Cases include the standard best-effort asynchronous delivery, preventing SPAM *etc.*; we extend to giving the recipient more control over message acceptance without allowing the misuse of inferring exact acceptance criteria. Our system, called *WebMail,* is a collection of web services that are orchestrated using the Business Process Execution Language (BPEL) [2].

In a similar effort, WSEmail [17] provides flexible means to communication, such as, dynamically discovering and negotiating communication protocols such as in Instant Messaging (IM), *etc*. AMPol [1] extends WSEmail by separating policies from delivery mechanisms, thereby achieving flexibility of operation. Our previous feedback-based recipient controlled email framework [12] extended traditional SMTP-based email flows, thereby alleviating some annoying misuse cases of earlier systems. In this work, we collect best of the two approaches, by designing a comprehensive web-based solution using standard methods.

The rest of the paper is organized as follows. Section 2 specifies Use cases enabled and Misuse Cases prevented in WebMail. Section 3 provides an overview of SMTP-based conventional email and possible transmission using Web Services. Section 4 specifies WebMail family of services using WSDL, and Section 5 presents their process integration using BPEL. Section 6 shows how our specification enables specified use cases and prevents described Misuse Cases. Section 7 ensures process integrity as a distributed system and Section 8 shows how inferring user preferences in mail acceptance criteria can be prevented. Section 9 describes related work and section 10 concludes the paper.

2. Use cases and Misuse cases

Known Use Cases enabled by conventional SMTP-based mail are as follows:

Use Case 1: *Best effort transmission of a text message from a sender (the principal actor) to a recipient (the secondary actor) through intermediate mail servers (auxiliary actors).*

Use Case 2: *Error reporting on transmission failure.*

A message transmission – broken down into three logical steps, is considered *complete* only if the message is routed to the recipient's mailbox. The steps are: from the sender to its email service providers (SESP); from SESP to recipients' email service provider (RESP); and finally from RESP to the recipient's mail box. However, physically, multiple mail servers may be involved and are subsumed under the logical entities – SESP and RESP. Message transmission may not be complete due to many failures, upon which the first point of failure detection is expected to inform the sender using another email message.

Specialization of Use Cases: Above two use cases can be specialized for a variety of message types and properties of transmission channels. Standard use cases supported by SMTP implementations are:

1. Best-effort transmission of enhanced content including text and MIME messages [5].
2. Enforcing security mechanisms such as transmitting authenticated message and using encrypted channel.
3. Best-effort transmission of message acknowledgements.

Best-effort relates to asynchronous transfer of messages across hosts on the internet, because recipient process may not be active when the sender process contacts them. In addition, SMTP ext- ensions [9, 18] include commands and replies for source auth- entication and negotiations for establishing a secure channel for synchronous transmission. Finally, SMTP also facilitates delivery receipts in the form of another email. These Use Cases are supp-orted by functionality built into communicating server processes.

Email delivery is subject to many misuse cases, including lack of source authentication, loss of privacy and integrity of content, receiving unsolicited commercial email (spam), email bombs [4], *etc*, of which our design prevents the following:

1. *Violating integrity and leaking or altering content*: Allowing unintended mal-actors to read message contents and alteration.
2. *Impersonating senders*: Allowing mal-actors to assume the identity of another person in a mail message.
3. *Email bombs*: This is a variation of DoS attack on email networks, where mail servers receive large number of messages, leading to denial of email service.
4. *Receiving undesirable email (spam)*: Allowing undesirable email to reach recipients' mailbox.

Although the STARTTLS [9] command exists in SMTP, most email messages are sent in clear-text over the wire, and stored as such at mail servers, thereby permitting the first misuse case to occur. Similarly, although the SMTP AUTH [18] exists, it requires prior exchange of secrets, which does not happen in most cases – thereby being subjected to the second misuse through sender-address spoofing. Current SMTP-like server designs result in being subjected to the third misuse case, where lack of recipient control over message delivery results in the last misuse case. Recent attention to spam has resulted in some proposals to add automated recipient controls to the message flow pipeline including, providing feedback about rejected messages [12]. A drawback of delivery controls is the inadvertent disclosure of acceptance criteria, that can now be used to defeat its purpose [13]. We also add this misuse case to the list of standard misuse cases and propose a solution in section 8.

3. Overview of message delivery

Figure 1 show a conventional email message originated by the principal actor (*i.e.*, sender) routed to the sender's email service provider (SESP) that transmits the message to the recipient's service provider (RESP). From here the recipient's mail agent picks up the delivered message [15]. Email service providers (ESPs) make this model of mail delivery possible, and thereby alleviating the senders and recipients to be online for synch-ronous message transfer. Also, they add important functionality, such as, filtering mail, virus scans, *etc.*, and enhance scalability by lumping messages destined to the same RESP to the same delivery attempt. Doing so brings the mail pipeline under the control of several auxiliary actors such as, reputation services (DCC [11], Cloudmark [10]) to check the ESP credibility or escrow services (for attention bonds [16]), *etc.* These interactions are represented by dotted double arrows in figure 1.

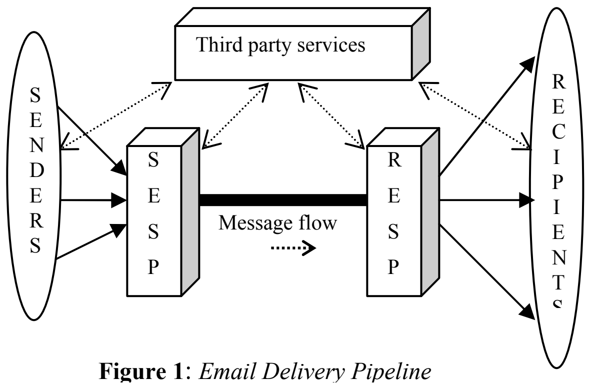

Figure 1: *Email Delivery Pipeline*

In a Web Services based message transmission, we replace each actor by one or more Web Services. Together these Web Services form a family referred to as the *WebMail family*. Here we show different orchestrations of these Web Services providing many *flavors* of email transmissions. We also show that earlier solutions for conventional systems can be readily adapted for the Web Services environment and possibly improved upon.

4. Web Services for Message Transmission

In this section we begin with the basic technical details of our model. Three basic components are considered for our specifications. First, we describe the types and parts of messages that are exchanged between Web Services. Then, we specify various Web Services that constitute the WebMail family. Finally, we specify various orchestrations of the WebMail family using abbreviated BPEL process specifications as done in [6].

4.1 Message Types

Message types define the protocol used for communication, *i.e.*, service interfaces are understood in terms of their input and output messages. Here, we limit the types of transported objects, however, our list is extensible and it is possible to include the complete set of MIME [8] objects. Basic types are described in table 1, and complex (*i.e.*, structural) types in table 2. We give these type definitions for completion. We don't intend to leverage on their type structure for the purposes of this paper. Our code (shown later) can be modified to be used with other typed structures as well. For instance, several techniques use custom structures for 'time' or 'credential', *etc.*, so we simply refer to them using an XML namespace element. For brevity, we omit the WSDL syntax for type definition. (For details, see [21]).

Table 1: *Basic types of message elements*

Type Name	Primitive Type	Example
MIME	ASCII string	Application/PDF
PKISignature	ASCII String	463hfd$&47654
Message ID	Long Int	239809832092
MType	Character string	Urgent, Personal, …
AckRqd	Boolean	Yes/No
Number	Positive Int	100
Nonce	Positive Int	10000
Email Address	ASCII string	abc@xyz.com
Password	ASCII string	******
Answer	ASCII string	Xy3

Table 2: *Complex types of message elements*

Type Name	Type Structure	Example
Time	XmlNS=URI#Time	10:00 A.M EST
Key Pair	IntXInt	(53,97)
Credential	XmlNS=URI#Cred	Credential struct
Image	XmlNS=URI#Jpeg	JPEG struct
AObject	Application/Type	PDF file
Credential Chain	Credential*	Cred1, …, CredN
Currency	Enum: {$, £}	$, £
Bond	XmlNS=URI#Bond	$3.5 Cred 1
Turing test	Image	10101..01,
Turing test reply	ImageXAnswer	(10101..01, xy3)
Content	String?, AObject*	"Example", Image

4.2 Messages

Message types (summarized in table 5) are described next. Structure of a *mail message* is presented first. This message contains routing information, objects to be transmitted and additional attributes that aid the delivery of the message. Message attributes are used by downstream processes to make routing decisions [12]. Mail message is described in WSDL format in listing 1. In the following listings character '*' signifies zero or more repetitions, '?' zero or one occurrence and '+' means one or more repetitions.

```
<message name="MailMessage">
  <part name="From" element="Email Address"/>+
  <part name="To" element="Email Address"/>+
  <part name="Date" element="Time"/>+
  <part name="ID" element="Message ID"/>+
  <part name="Surety" element="Bond"/>?
  <part name="Pass" element="Password"/>*
  <part name="Ack" element="AckRqd"/>*
  <part name="Sign" element="PKISignature"/>*
  <part name="RTT reply" element="Turing
                             Test Reply"/>*
  <part name="MType" element="String"/>?
  <part name="Subject" element="String"/>?
  <part name="Body" element="Content"/>?
</message>
```

Listing 1: *WSDL Mail Message*

In addition to mail messages, clients and servers transmit several other types of messages – enable underlying communication protocols by informing the status of the communication, properties of the transmission (QoS,) *etc*. Table 3 and 4 show their (abbreviated) WSDL syntax.

Table 3: *WSDL Application Data*

Message Type	Part , Multiplicity	Part type
ReceiptNotice	"Date" + "ID" + "Sign"*	"Time" "Message ID" "PKISignature"
FailNotice	"Date" + "ID" + "Error"+ "Sign"*	"Time" "Message ID" "Character string" "PKISignature"
RejectNotice	"Date" + "ID" + "Eval Policy"+ "Sign"*	"Time" "Message ID" "Policy" "PKISignature"
RefinementMsg	"Date" + "ID" + "Surety"* "Sign"* "MType" ? "RTT"* "Body"*	"Time" "Message ID" "Bond" "PKISignature" "Character string" "Turing Test" "Content"
RefinementFailure	"ID" + "RError"+	"Message ID" "Character string"
InformationMsg	"Date" + "ID" + "Information"+ "Sign"*	"Time" "Message ID" "Character string" "PKISignature"

Table 4: *WSDL Control Data*

Message Type	Part, Multiplicity	Part type
MailIntent	"Date" + "NoOfMsgs " + "Sign"*	"Time" " Number " "PKISignature"
SLA	"Date" + "AllowedNo." + "Sign"*	"Time" "Number" "PKISignature"
PKICertificate	" Key " + " Session "*	" Credential " " Nonce "
AcceptancePolicy	"Date" + "Surety"* "Sign"* "MType" ? "RTT"* "Body"*	"Time" "Bond" "PKISignature" "Character string" "Turing Test" "Content"

Table 5: *Types of messages and their utility*

Message Type	Utility
Mail Message	Message to be delivered
Receipt notice	Notice of receipt and acceptance for delivery of
FailNotice	Notice of delivery failure
RejectNotice	Notice of delivery rejection
RefinementMsg	Changes desired in a mail message
RefinementFailure	Desired changes not possible
InformationMsg	Third party message evaluations
MailIntent	Indication of transmission intent
SLA	QoS for invocations
AcceptancePolicy	Acceptance rules advertisement
PKICertificate	Proof of identity and data secrecy

Message definitions in table 3 determine the application data or the payload for the message communications. Table 4 defines protocol data exchanged for effectively completing the task at hand. In particular, *Mail Intent*, message expresses the intent to send messages, *SLA* message is a response to mail intent message indicating number of messages allowed; while *Acceptance Policy* message states acceptable message attributes.

4.3 WSEmail family of Web Services

Next, we design a family of Web Services that perform various tasks to aid delivery of email messages. We list the set of externally callable methods for each principal involved in message delivery.

Sender's ESP (SESP): Sender's email service provider is designed to receive messages, route them to the destination, examine and *repair messages* [12] before sending them, refine messages rejected by RESP [12], *etc.*

1. SESPConnectPT
2. SESPReceiveMsgPT
3. SESPAuthPT
4. SESPDeliveryPT
5. SESPMsgCallbackPT
6. SESPImprovementPT
7. SESPVirusExaminationPT
8. SESPVirusRemovalPT

Sender: Sender's may need to expose a callback interface to receive rejection notices or notices for improving messages

1. SenderMsgCallbackPT
2. SenderMsgRefinementPT
3. SenderPasswdCallbackPT

Recipient's ESP (RESP): Recipient's ESP provides the following set of services.

1. RESPHeloPT
2. RESP-TLSPT
3. RESPReceiveMsgPT
4. RESPVirusScanPT
5 RESPFilterPT
6. RESPControlPT
7. RESPSanitizationPT
8. RESPDeliveryPT
9. RESPStoragePT
10. RESPImprovementPT

Recipient: A recipient need not expose any service; however, some recipients may allow their service providers to "push" messages to the recipient's host through *RReceiveMsgPT.*

In addition to SESP and RESP services, third party services may be invoked during message transmission. Here we consider only two Web Services, though this list could easily be extended.

Third party services: *CheckSumPT* can be invoked to verify if a message is a bulk message and *BondVerificationPT* verifies the authenticity of an attached monetary bond.

Tables 6, 7 and 8 describe the Web Service portypes.

Table 6: *portTypes for SESP services*

PortType	Input	Output	Fault(s)
SESPReceiveMsgPT	MailMessage	ReceiptNotice	FailNotice
SESPConnectPT	MailMessage	IntentMessage	SLAFail
SESPAuthPT		PKICertificate	FailNotice
SESPDeliveryPT	MailMessage	ReceiptNotice	FailNotice
SESPCallbackPT	RefinementMsg		FailNotice
SESPImprovemntPT	RefinementMsg	MailMessage	FailNotice
SESPExaminationPT	MailMessage	InformatnMsg	
SESPVRemovalPT	MailMessage	MailMessage	

Table 7: *portTypes for RESP services*

PortType	Input	Output	Fault(s)
RESPHeloPT	MailIntent	SLA	
RESP-TLSPT	PKICertificate	PKICertificate	FailNotice
RESPReceiveMsgPT	MailMessage	ReceiptNotice	RejectNtc
RESPVirusScanPT	MailMessage	InformatnMsg	TimeOut
RESPFilterPT	MailMessage	InformatnMsg	TimeOut
RESPControlPT	Sender	InformatnMsg	
RESPSanitizatnPT	MailMessage	MailMessage	TimeOut
RESPDeliveryPT	MailMessage	ReceiptNotice	FailNotice
RESPStoragePT	MailMessage		FailNotice
RESPImprovmtPT	MailMessage	RefinementMsg	TimeOut

Table 8: *portTypes for third party services*

PortType	Input	Output	Fault(s)
CheckSumPT	MailMessage	InformationMsg	TimeOut
bondVerificatPT	MailMessage	InformationMsg	TimeOut

5. BPEL Orchestration of WSEmail

In this section, we begin with a basic set of synchronized Web Service invocations for mail delivery. We illustrate typical activities, in the notation borrowed from BPEL specification manual by Andrews, Curbera [1], *et al*. SESP is described in figure 2 and RESP in figure 3, followed by their (abbreviated) process descriptions (resp. listings 2 and 3).

5.1 SESP Process specification

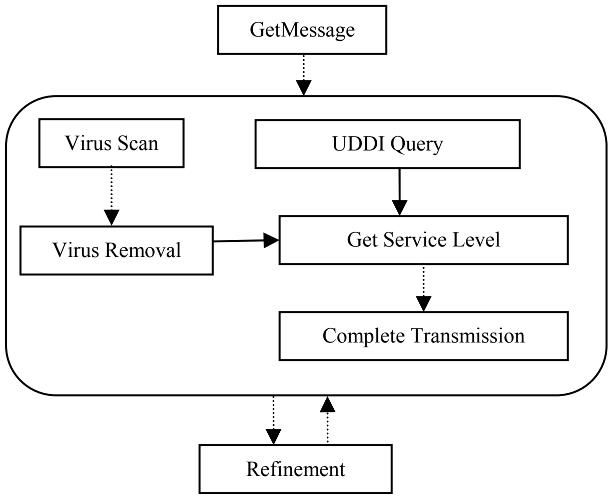

Figure 2: An *SESP Orchestration*

Dotted lines in figure 2 (and 3) indicate *sequential executions* and solid lines indicate *control dependencies for synchronizing concurrent activities*. Note that the diagram does not give details about exception handling. These cases are showcased in code later; and are ignored here for the sake of clarity. In figure 2, Senders invoke SESP's ReceiveMsgPT, the initial activity. Next, SESP process initiates two concurrent threads of execution, *viz.*, virus scan of message and UDDI location of RESP. The virus removal process is run on infected messages. Finally, the SESP invokes the *HeloPT* and *RecieveMsgPT services* of the RESP to begin message delivery.

```
1  sequence
2  | flow
3  | | sequence //New message from sender
4  | |   receive "SESPReceiveMsgPT(M)"
5  | | sequence //Refined message retransmn
6  | |   receive "SESPReceiveMsgPT(M)"
7  | | sequence // Call back service
8  | |    receive "SESPCallbackPT(RefM)">
9  | |    "M" invoke "SESPImprovementPT(RefM)"
10 | |          throw "FailureFault(RefM)"
11 | |     reply "SESPReceiveMsgPT(M)"
12 | flow // message preparation
13 |  | links
14 |  |     "fix-deliver"
15 |  |     "UDDI-resn"
16 |  | sequence
17 |  |   "Rlt" invoke "SESPExaminationPT(M)"
18 |  |   switch
19 |  |    case condition="Rlt=true"
20 |  |      "M" invoke SESPVirusRemovalPT(M)"
21 |  |      source link="fix-deliver"
22 |  |    otherwise
23 |  |         empty // do nothing
24 |  | sequence // where to send message?
25 |  |   "IP" invoke "UDDIService(TO)"
26 |  |     source link="UDDI-resn"
27 |  | sequence // send message to RESP
28 |  |   "SLA" invoke "SESPConnectPT(M)"
29 |  |     target link="UDDI-resn"
30 |  |     target link="fix-deliver"
31 |  |   // begin delivery
32 |  |   while condition="number < SLA"
33 |  |     flow
34 |  |     sequence
35 |  |       "R" invoke "SESPDeliveryPT(M)"
36 |  |         catch "RejectionFault"
37 |  |          reply SenderMsgCallbackPT(N)"
```

Listing 2: Example SESP Process

Listing 2 shows a typical SESP process in BPEL syntax. The code has three main blocks: message reception (line 3–6); message preparation (lines 12–26); and message delivery (lines 27–37). The first part accepts messages from a sender, to be delivered to some recipient. In addition, the SESP process allows its message callback service to retransmit an earlier rejected (but now revised) message. In other words, messages rejected earlier, say for lack of authentication or other attributes desired by RESP, are *repaired* with the help of this feedback. Next, each message enqueued for delivery is subject to checks (like virus scan, *etc.*) to ensure quality of a message. Finally, the message is sent across to the RESP.

5.2 RESP Process specification

Next, we define an RESP process that enforces a sample service level agreement (SLA) and a *reasonable* message acceptance policy (AP), given below.

SLA	*Allow*		*10 messages per connection*
	Allow		*Feedback for rejected messages*
AP	*Accept message*	*IF*	*No virus/worm is attached*
		AND	*Filter allows receipt*
			OR
			Distributed checksum allows receipt
	Accept message	*IF*	*No virus/worm is attached*
		AND	*Message bonded with value > b*
		AND	*Bond is verified by an escrow service*

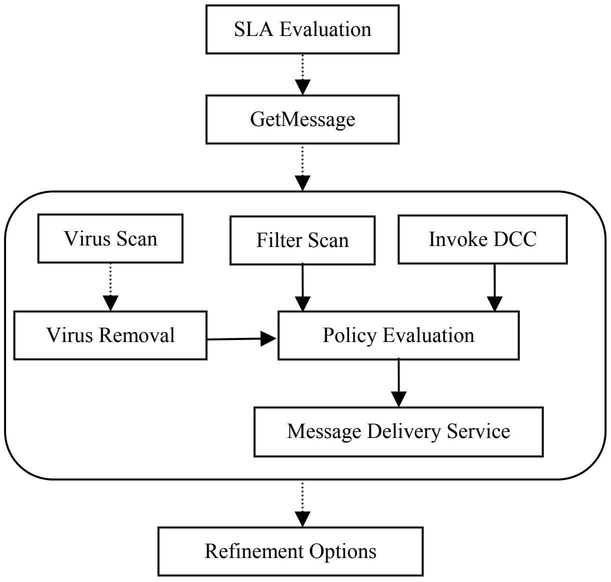

Figure 3: An *RESP Orchestration*

As shown in figure 3, upon invocation of RESP's RecieveMsgPT the message is transmitted to RESP. For each received message, the RESP applies a message acceptance policy to accept or reject it. If the transmitted mail fails to satisfy this policy, the RESP either returns a *rejection notice* or a *refinement message*. The refinement message suggests changing some parts of the message that may make it acceptable to the RESP, while rejection notice is a permanent rejection. As a result, refinement activity may begin at the SESP. Note that based on its own policy, an SESP may decide to ignore all advice, and consequently, the callback service interface may not be exposed (the current strategy in existing SMTP implementations). On the other extreme, if neither party stops the refinement process, it may go on forever. Many such strategies have been studied by researchers in other contexts (like automated trust negotiation [20], *etc.*), and can be supported here. In the code presented next, we restrict refining a message up to a fixed number of times (5). This is because we haven't seen the need yet for a more complex strategy.

The RESP process is made up of four main parts, as shown in listing 3, *viz*, message reception from SESP (lines 3—5); invocation of helper services to gauge *message quality* (lines 7—23); acceptance policy evaluation based on message quality (lines 24—47) and finally, computing feedback for rejected messages (lines 49—57). The RESP waits for messages to arrive, and if they satisfy the service level agreement (SLA), they are accepted (as shown in listing 3). Next, the RESP makes concurrent calls to several 'helper' services, like Bayesian filtering service, bond verification service, distributed checksums, virus scans, *etc.*, to gauge the quality of an incoming message. Each service evaluates a message and reports its findings to the RESP process in an information message. On their termination, the RESP process starts evaluating the concerned message based on RESP's acceptance policy and evaluations by helper services. During this stage a message may be accepted or rejected. Rejected messages may be returned to the SESP with feedback on rejection. SESP (and sender) can then retry transmission after making changes to the message such that it satisfies RESP's acceptance policy.

```
1  sequence
2    // logic for generating SLA
3    switch // Evaluate SLA
4      case condition="number &lt; 11"
5        receive "RESPReceiveMsgPT(M)"
6    flow // Invoke concurrent processes
7      sequence // Virus scanning
8     "Rlt" invoke "RESPExaminationPT(M)"
9       switch
10         case condition="Rlt=true"
11           "M" invoke "RESPVirusRemovalPT(M)"
12         otherwise
13           empty
14               source link="empty"
15     sequence // Distributed checksum
16      "checksumOK" invoke "CheckSumPT(M)"
17               source link="dcc-deliver"
18     sequence // Verify bond
19      "V" invoke "bondVerificationPT(M)"
20               source link="bond-verify"
21     sequence> // Bayesian filtering
22      "filterOK" invoke "RESPFilterPT(M)"
23               source link="filtering"
24     <!— enforcing acceptance policy -->
25     sequence
26     switch
27       case condition="(fixed OR empty)
28          AND (checksumOK OR filterOK))"
29         "N" invoke "RESPStoragePT(M)"
30          switch
31            case condition="N==ReceiptNotice"
32            reply "SESPDeliveryPT(N)"
33            case condition="N = FailNotice"
34                  throw "FailFault"
35            otherwise empty // do nothing
36       case condition="(fixed OR empty) AND
36             (verified AND bond &gt; b)">
37         "N" invoke "RESPStoragePT(M)"
38         switch
39         case condition=" N = ReceiptNotice"
40           reply "SESPDeliveryPT(N)"
41         case condition="N = FailNotice"
42                 throw "FailFault"
43         otherwise
44                 empty // do nothing
45        case condition="NOT fixed OR NOT
46              (checksumOK AND filterOK)>
47              throw "RejectionFault"
48        otherwise
49          sequence
50          switch
51            case condition="history &gt; 5"
52            // maximum invocations = 5
53            "RM"invoke "RESPImprovementPT(M)"
54            // store M's refinement history
55            invoke "SESPCallbackPT(RM)
56            otherwise
57               empty // do nothing
```

Listing 3: Example RESP Process

Example 1: *Assume a mail message (M) that contains the following appropriately initialized parts: From, To, Date, ID, Subject and Body. We make the following assumptions:*

- *M does not contain any attached virus/worm*
- *M is the only message in queue*
- *RESP's SLA accepts 10 messages per connection, and provides feedback for rejected messages.*
- *Acceptance policy requires that no virus be attached to a message, and either the message has a bond ("Surety") or satisfies the Bayesian filter.*
- *Message content may contain prohibited words.*

According to the RESP described in listing 3, with the change that above policy instead of the one shown in table 4 is evaluated, M will not be accepted for delivery at the RESP (lines 24—47). This is because it fails to satisfy both conditions – it doesn't include a valid bond and it doesn't satisfy the Bayesian filter on account of the prohibited words in its body. Next, (lines 51—54) the RESP process initiates a call to the message improvement service (to allow the sender to revise the message). The content of the refinement message would include the following parts: Date, ID, Sign, Surety and Body – the missing information that caused rejection. Essentially, this response provides the sender acceptable values for the parts Date, ID, Surety and Body. That is, the refinement message identifies the deficiencies in M: no valid bond (or surety) and presence of prohibited words. Once made aware, the sender may choose to alter the rejected message, so that it reaches its destination [12].

6. Coverage of use cases and misuse cases

We show next that the set of Web Service definitions, identified above, satisfy all stated use cases and avoid all mis-uses. We give our arguments in the form of (abbreviated) BPEL specifications as a proof of our claims.

6.1 Coverage of standard use cases

Sender invokes SESP's message delivery operation in line 4 – listing 2 (resp. SESP invokes RESP's delivery operation in line 5 – listing 3). Input messages of type text or MIME messages (identified in the type declarations – see [21]) are *queued for delivery*. The SESP service interface (resp. RESP interface) provides only best-effort delivery. As a result, if delivery fails at this stage, an error is generated – line 36, listing 2 (resp. line 42 listing 3). If all prerequisites for delivery are satisfied, then both SESP and RESP processes are guaranteed to attempt delivery. (Note, that the listings include only one delivery attempt, but multiple delivery attempts can be supported). Hence, the SESP and the RESP processes satisfy both the requirements of standard use cases – best effort transmission and error report on delivery failure. Consequently, the services defined here are sufficient for supporting standard use cases; additional proof is provided next.

Use Case: Authenticated message transmission

This use case is supported through invocations of the SenderPasswdCallbackPT and SESPAuthPT services.

SESP process modification

```
Sequence
  Receive Message M
  Invoke SenderPasswdCallbackPT
  Switch
    Case: Password is correct
      ... // proceed to other delivery tasks
    Otherwise
      Throw <Failure Fault, message: incorrect password>
```

RESP process modification

```
Sequence
  Receive RESPHeloPT
  Receive Message M
  Invoke SESPAuthPT
  Switch
     Case: Credential verified
              ... // proceed to other delivery tasks
     Otherwise
        Throw <Failure Fault, message: invalid credential>
```

Code example above illustrates a simple (and scalable) way to support authenticated messages. Here, messages are authent-icated in two tiers, *i.e.*, message senders are authenticated by their SESPs; while SESP is authenticated (using AuthPT service) by the RESP. It should be noted that this strategy provides only partial guarantees to sender authentication (since the sender is never directly authenticated by the RESP). More elaborate schemes, like, PKI or secret key schemes like Kerberos are also possible, though we don't specify them here.

Use Case: Secure message transmission
This use case is supported through successive invocations of the RESP-TLSPT

RESP Process modification

```
Sequence
  Receive RESPHeloPT
  Invoke RESP-TLSPT
  Switch
  Case: while SLA
     Receive Message M
        ... // proceed to other delivery tasks
  Otherwise
        Throw <Failure Fault, message: not allowed>
```

At each successive hop of a message, the sending agent can invoke transmission over TLS (or SSL) for privacy and integrity of data over the wire. This use case completes the set of standard use cases for email delivery.

6.2 Preventing misuse cases

Here we show that the set of Web Services we define are adequate for preventing stated misuse cases. Again, we show coverage of all misuse cases with abbreviated BPEL specifications. We use listings 2 and 3 to give informal proof sketches of our claim. In addition, misuse cases like integrity, privacy, non-repudiation of message initiation are dependent upon more basic misuses like lack of sender authentication and absence of secure transmission. So, next we show how basic misuses prevented, rather than the ones dependent on them.

Misuse Case 1: Denial of Email Service (email bombs)
This misuse is prevented using service level agreement for incoming mail connections. For instance, a service level agreement (SLA) can restrict number of concurrent connections from a particular domain and number of messages transmitted per connection (for instance, listing 3, line 4 restricts an SESP to only 10 messages per connection).

Misuse Case 2: Transmission in clear-text with no sender authentication
These misuses are prevented using acceptance policies for incoming messages. For instance, an acceptance policy requiring messages be authenticated and transmitted over a secure channel is easily encoded in BPEL as:

RESP Process modification

```
...
  Switch
     Case: "Password=correct AND channel= encrypted"
       Rnotice=  Invoke RESPStoragePT(Msg)
     Otherwise
        RMsg = Invoke RESPImprovmentPT(Msg)
        Reply SESPCallBack(RMsg)
  End switch
---
```

Consider lines 24 onwards in listing 3, where messages attributes are evaluated by the acceptance policy for the delivery session. The above policy that checks for password based authentication and encrypted channel can be applied *in conjunction* with other message acceptance requirements. That is, prevention of this misuse is possible by enforcing the correct acceptance policy.

Misuse Case 3: Controlling unwanted messages
Similar to the prevention of misuse case 2, this misuse is prevented using acceptance policies. The difference with the previous case is in the invocation of different Web Services like (FilterPT, DCC, *etc.*) during acceptance policy evaluation. For instance, a policy that requires the Bayesian filter and checksum service to approve a message is coded in BPEL as follows:

RESP Process modification

```
...
  Switch
     Case: Filter = false && checksum = false
       Rnotice=  Invoke RESPStoragePT(Msg)
     Otherwise
        RMsg = Invoke RESPImprovmentPT(Msg)
        Reply SESPCallBack(RMsg)
  End switch
---
```

As before, these conditions can be enforced *in conjunction* with other conditions (or otherwise) in listing 3 (line 24 onwards).

7. Ensuring processes integrity

In this section we analyze SESP and RESP processes and informally argue that they exhibit several desirable properties. SESP and RESP processes include synchronized and parallel invocations of Web Services. For correctness of these calls, we show that the processes possess.

Deadlock freedom [3]: This property states that parallel invocations of Web Services are independent of each other, i.e., they do not block while waiting for the other to terminate or release a lock on synchronized resources.
Interference freedom [3]: This property states that execution of atomic steps of one component never falsify the properties enabled because of another component.
Distributed Termination [3]: This property states that a process terminates or stops executing after a finite amount of time.

Because of space limitations, we give informal arguments. Work on formal proofs is in progress. In the following analysis, we categorize pairs (or sets) of programs along the following terms:

Parallel but disjoint [3]: A pair of programs is considered parallel but disjoint if one program cannot change variables accessed by other program.

Parallel with shared variables [3]: A pair of programs is parallel with shared variables if any one program can change variables accessed by the other.

Parallel with shared variables and synchronization [3]: Parallel programs with shared variables are also synchronized if they are able to suspend their execution while waiting on another program component to finish executing.

Before we begin arguing about the properties of our implementation of SESP and RESP processes, we give the abbreviated BPEL specification of the sender process.

Sender Process

```
Declarations: process, variables, faults
  Flow
    Invoke SESPReceiveMsgPT(M)
    Receive SenderCallbackPT
      Sequence
        // improve message
        invoke SESPReceiveMsgPT(M)
```

Note that Sender, SESP and RESP processes fall in the first category stated above (parallel, disjoint processes). Also, we assume that individual Web Service components that are disjoint and recursion free and *always* satisfy their contracts. That is, assuming that their preconditions are met, they always terminate satisfying all their post conditions – their fault model is not included. Since process specifications do not involve asynchronous invocations, self recursion, and unbounded mutual recursion; therefore, following properties follow easily.

Proposition 1: Following properties of processes hold

1. Sender process exhibits deadlock freedom and interference freedom.
2. Sender process terminates.
3. SESP message transmission process is interference free and terminates
4. The SESP process is deadlock free.
5. The RESP process is deadlock free.
6. RESP message transmission process is interference free and terminates

Proof: See [21]

8. Privacy Leakages due to Feedback

Example 1 shows that providing feedback not only reveals the policy that is being evaluated at the RESP to the sender, but also leaks several other types of information. For instance, in example 1, the sender could determine the expressions rejected by the RESP's Bayesian filter. This information can be misused by the sender to send undesira-ble messages to the recipient by simply camouflaging the `flagged' expressions – using HTML tags, insertion of spaces and other similar techniques. Other types of leakages [13] that compromise recipient's private information are also possible.

Leakages are categorized into two classes [13], *viz*, those due to feedback provided in-band with the transmission channel, and those due to out of band feedback channels. In the case of example 1, the leakage of information occurs due to in band feedback channel. These can by simply prevented in the SLA by prohibiting feedback. Consequently, the message improvement service will not be invoked. However, leakage is still possible, as shown next. Consider a scenario where an acceptance policy requires that a message satisfy the Bayesian filter *and* include a valid bond. Because of this policy whenever the bond is seized by a recipient, causing out of band monetary flow, it reveals the *strength* of the filter to the sender as the sender gets the confirmation that the message satisfied the Bayesian filter. Clearly, strength of the filter is sensitive information that must be protected, as argued above.

In [13] we develop methods for preventing out of band privacy leakages. These are directly applicable to the BPEL processes described here. We translate their solution for logic programs to our imperative programs. In addition, we show how process synchronization can be used to enforce their solution, a study missing in earlier work. First, we illustrate the problem with an original (unsafe) policy and its BPEL specification.

Policy 1 [Original (Unsafe) Policy]: *Consider the following acceptance policy for accepting messages:*

Accept	*IF*	*Sender is not blacklisted and bond* $\geq a$
message	*OR*	*Sender blacklisted and bond* $\geq b$ $(b>a)$

As shown earlier[13], this is an unsafe policy since it introduces an out of band feedback channel. For instance, if a sender sends a message bonded with value $c \in (a,b)$ *and the bond is seized, then money transfer indicates to the sender that he or she is not blacklisted by the particular recipient. BPEL specification of this policy enforcement is as follows:*

Policy evaluation block in RESP process

```
Sequence
  Switch
    Case: Sender ∉ blacklist AND bond > a
      Rnotice= Invoke RESPDeliveryPT(Msg)
    Case: Sender blacklist AND bond > b
      Rnotice= Invoke RESPDeliveryPT(Msg)
    Otherwise
      RMsg = Invoke RESPImprovmentPT(Msg)
      Reply SESPCallBack(RMsg)
```

8.1 Policy Transformation

Out of band leakages described above are harder to prevent without discontinuing the use of Web Services that introduce the leakage channel. That is, protection against privacy leakages requires that recipients and RESPs disable the use of such Web Services. However, this condition is too strict; an alternate solution exists that achieves the same goal without requiring the recipients to write truncated acceptance policies. This is done be automatically generating two safe policies from the original: the *necessary* and the *sufficient* policy.

Intuitively, the necessary policy is a weaker policy (truncated form of original policy) that does not invoke *leaky* Web Services. On the other hand, the sufficient policy is a strictly stronger policy that does not invoke leaky Web Services. With the ability to automatically construct these policies, a policy author can still enforce the original policy with a trusted client; and use the necessary and sufficient policies in tandem with a suspicious or an unknown client. Their construction and use is detailed next.

For the transformation procedures below we assume that a policy can be represented as a logical formula in disjunctive normal form (DNF), *i.e.*, it can be represented as $d_1 \vee d_2 \vee \ldots \vee d_n$ where each d_i is a conjunction of Boolean conditions.

```
NecessaryTransform(Policy, private):
Input: A set of policy rules
Input: A set of sensitive information attributes
Output: A set of policy rules that protect sensitive information
    if (Policy rules contains $p \in private$)
      Repeat till Policy does not contain any $p \in private$
1.      choose a rule $\in$ Policy | rule= $\vee_i d_i$ and some $d_i$ contain p
2.      modify each such $d_i$ such that it does not contain p
    else
        return
```

Policy 2 [Necessary Policy]: *Consider the original policy, discussed in Policy 1.*

Accept	*IF*	*Sender is not blacklisted and bond $\geq$ a*
message	*OR*	*Sender blacklisted and bond $\geq$ b (b>a)*

Applying the NecessaryTransform procedure to the original unsafe policy yields the following necessary policy:

Accept message	*IF*	*Bond $\geq$ a*

In this particular example, the contents of a blacklist are considered sensitive. Consider the evaluation of this policy at RESP:

Policy evaluation block in RESP process

```
Sequence
  Switch
    Case: bond > a
      Rnotice= Invoke RESPDeliveryPT(Msg)
    Otherwise
      RMsg = Invoke RESPImprovmentPT(Msg)
      Reply SESPCallBack(RMsg)
```

As is evident from the code above, this policy accepts messages with a minimum bond value, and assuming recipient will seize bonds for all unwanted messages, the only information that this policy leaks is that the recipient requires a bond value of a for messages to be accepted. No information about content of recipient's blacklist can be deduced.

```
SufficientTransform(Policy, private):
Input: A set of policy rules
Input: A set of sensitive information attributes
Output: A set of policy rules that protect sensitive information
    if (Policy rules contains $p \in private$)
      Repeat till Policy does not contain any $p \in private$
1.      choose a pair of rules $\in$ Policy | rule1= $\vee_i d_i$ and some $d_i$ contain
p and rule2= $\vee_j D_j$ and some $D_i$ contain NOT(p)
2.      remove rule1 and rule2 and construct a new rule such that
              rule=$(\vee_i d_i) \vee (\vee_j D_j)$ except the disjuncts containing p
    else
        return
```

Policy 3 [Sufficient Policy]: *Consider the original policy, discussed in Policy 1. Applying the SufficientTransform procedure to the original unsafe policy yields the following necessary policy:*

Sufficient policy:

Accept message	*IF*	*Bond $\geq$ b*

Consider the evaluation of this policy at RESP:

Policy evaluation block in RESP process

```
Begin Sequence
  Switch
    Case: bond > b
      Rnotice= Invoke RESPDeliveryPT(Msg)
    Otherwise
      Throw RejectFault(Msg)
  End switch
End Sequence
```

As in the previous case, the sufficient policy enforcement can only reveal to the sender that the message requires a minimum bond value of b. No information about the contents of the blacklist is divulged.

9. Related Work

Lux, May, *et al* in [17] introduce WSEmail, *i.e.*, transmission of messages using Web Services. Web Services lend additional flexibility to the message transmission process, while avoiding standard pitfalls, like, lack of sender authentication, susceptibility to spam, *etc.* However, details regarding the standard SMTP use cases are missing, as well are the details on orchestration of related Web Services. Here we fill these gaps.

Next closely related work is by Afandi [1], where the author discusses *adaptive* policies for messaging systems (like WSEmail). The central idea is to separate policies from the mechanism to allow flexibility in the behavior of network components involved in message transmission; however, this work restricts to the design and architecture of the system. Here, we complement AMPol by a simple implementation using BPEL. Additionally, we provide sufficient evidence that the alluded misuse cases (in [1]) will be prevent by our orchestration.

Kaushik, Winsborough *et al* in [12, 13] solve similar problems in conventional systems, and provide several alternative solutions. We consider the applicability of their solutions, appropriately tailored, to the new domain. In addition, we show how process synchronization is used to enforce their solution, the piece missing in all earlier works. Finally, we give informal proofs of correctness of our implementation that uses parallel concurrent process for achieving message transmission.

Chafle, Chandra *et al* [6] present an analysis for decentralized orchestration of Web Services using BPEL. Though the problem we consider here is not directly related, but our analysis takes a leaf out of their synchronization analysis of BPEL orchestration.

10. Conclusion

In this paper we have analyzed an emerging Web Services based application for internet messaging known as WSEmail and compared it to the conventional messaging systems. Since the existing specifications for WSEmail don't consider all the standard use cases of current message delivery infrastructure or the set of misuse cases that must be prevented, we augment their architecture with our additions. We provide a formal specification of each Web Service considered and show that standard use cases are supported with the family of Services we have identified; and all misuse cases can be prevented with the same (extensible) set. We show how to orchestrate this family of services securely to achieve the goal of secure transmission of email messages, with no privacy leakages, a piece missing in most other works. In addition, we prove correctness of our specification.

11. REFERENCES

[1] R. N. Afandi, *AMPOL: Adaptive Messaging Policy Based System*, *Master's Thesis in Computer Science*, University of Illinois at Urbana-Champaigne, 2005.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, *Business Process Execution Language for Web Services*, 2003.

[3] K. R. Apt and E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1997.

[4] T. Bass, A. Freyre and D. Gruber, *E-Mail Bombs and Countermeasures:Cyber Attacks on Availability and Brand Integrity*, IEEE Network, 12 (1998), pp. 10--17.

[5] N. Borenstein and N. Freed, *RFC 1521 - MIME (Multipurpose Internet Mail Extensions)*, 1993.

[6] G. Chafle, S. Chandra, V. Mann and M. G. Nanda, *Decentralized Orchestration of Composite Web Services*, *Thirteenth international world wide web conference (WWW 2004)*, 2004.

[7] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, 2001.

[8] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions*, *RFC 2045*, 1996.

[9] P. Hoffman, *SMTP Service Extension for Secure SMTP over Transport Layer Security*, *RFC 3207*, 2002.

[10] http://www.cloudmark.com/, *Cloudmark*.

[11]http://www.rhyolite.com/anti-spam/dcc/, *Distributed Checksum Clearinghouse*.

[12] S. Kaushik, W. Winsborough, D. Wijesekera and P. Ammann, *Email Feedback: A Policy-Based Approach to Overcoming False Positives*, *3rd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FMSE 2005)*, Fairfax, VA, 2005, pp. 73--82.

[13] S. Kaushik, W. Winsborough, D. Wijesekera and P. Ammann, *Policy Transformations for Preventing Leakage of Sensitive Information in Email Systems*, in E. Damiani and P. Liu, eds., *20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Springer Berlin / Heidelberg, Sophia Antipolis, France, 2006.

[14] J. Klensin, *Simple Mail Transfer Protocol*, *RFC 2821*, 2001.

[15] J. F. Kurose and K. W. Ross, *Computer Networking : A Top-Down Approach Featuring the Internet*, Addison Wesley, 2004.

[16] T. Loder, M. V. Alstyne and R. Walsh, *An Economic Answer to Unsolicited Communication 5th ACM conference on Electronic Commerce*, 2004, pp. 40-50.

[17] K. D. Lux, M. J. May, N. L. Bhattad and C. A. Gunter:, *WSEmail: Secure Internet Messaging Based on Web Services*, *2005 IEEE International Conference on Web Services (ICWS 2005)*, Orlando, FL, 2005, pp. 75-82.

[18] J. Myers, *SMTP Service Extension for Authentication*, *RFC 2554*, 1999.

[19] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.

[20] T. Yu, X. Ma and M. Winslett:, PRUNES: an efficient and complete strategy for automated trust negotiation over the Internet. , 7th ACM Conference on Computer and Communications Security (CCS '00), Athens, Greece, 2000, pp. 210-219.

[21] S. Kaushik, D. Wijesekera and P. Ammann, *BPEL Orchestration of Secure WebMail*, Technical Report ISE-TR-06-08, George Mason University, Fairfax, VA, August 2006.