All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces

Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk
Chair for Network and Data Security
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
firstname.lastname@rub.de

Nils Gruschka
NEC Europe Ltd.
Heidelberg, Germany
gruschka@neclab.eu

Luigi Lo Iacono
Faculty of Information, Media and Electrical Engineering
Cologne University of Applied Sciences, Germany
luigi.lo_iacono@fh-koeln.de

ABSTRACT

Cloud Computing resources are handled through control interfaces. It is through these interfaces that the new machine images can be added, existing ones can be modified, and instances can be started or ceased. Effectively, a successful attack on a Cloud control interface grants the attacker a complete power over the victim's account, with all the stored data included.

In this paper, we provide a security analysis pertaining to the control interfaces of a large Public Cloud (Amazon) and a widely used Private Cloud software (Eucalyptus).

Our research results are alarming: in regards to the Amazon EC2 and S3 services, the control interfaces could be compromised via the *novel* signature wrapping and advanced XSS techniques. Similarly, the Eucalyptus control interfaces were vulnerable to *classical* signature wrapping attacks, and had nearly no protection against XSS. As a follow up to those discoveries, we additionally describe the countermeasures against these attacks, as well as introduce a novel "black box" analysis methodology for public Cloud interfaces.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized Access

General Terms

Security

1. INTRODUCTION

The cloud computing paradigm has been hailed for its promise of enormous cost-saving potential. In spite of this euphoria, the consequences regarding a migration to the cloud need to be thoroughly considered. Amongst many obstacles present, the highest weight is assigned to the issues arising within security [14].

Cloud security discussions to date mostly focus on the fact that customers must completely trust their cloud providers with respect to the confidentiality and integrity of their data, as well as computation faultlessness. However, another important area is often overlooked: if the Cloud control interface is compromised, the attacker gains immense potency over the customer's data. This attack vector is a novelty as the result of the control interface (alongside with virtualization techniques) being a new feature of the Cloud Computing paradigm, as NIST lists *On-demand self-service* and *Broad network access* as essential characteristics of Cloud Computing systems[1].

In this paper, we refer to two distinct classes of attacks on the two main authentication mechanisms used in Amazon EC2 and Eucalyptus cloud control interfaces. The first class of attacks complies of the *XML Signature Wrapping attacks* (or in short – *signature wrapping* attacks) [22, 12] on the public SOAP interface of the Cloud.

We demonstrate that these control interfaces are highly vulnerable to several new and classical variants of signature wrapping. For these attacks, knowledge of a single signed SOAP message is sufficient to attain a complete compromization of the security within the customer's account. The reason for this easiness is that one can generate arbitrary SOAP messages accepted by this interface from only one valid signature. To make things even worse, in one attack variant, knowledge of the (public) X.509 certificate alone enabled a successful execution of an arbitrary cloud control operation on behalf of the certificate owner. Those included actions such as starting or stopping virtual machines, downloading or uploading virtual machine image files, resetting the administrator's password for cloud instances, and so on.

The second class are advanced XSS attacks on browser based Web front-ends. We found a persistent Cross Site Scripting (XSS) vulnerability that allowed an adversary to perform an automated attack targeted at stealing username/ password data from EC2/S3 customers. This attack was made possible by the simple fact the Amazon shop and the Amazon cloud control interfaces share the same log-in credentials, thus any XSS attack on the (necessarily complex) shop interface can be turned into an XSS attack on the cloud control interface. The Eucalyptus Web front-end was equally prone to these kind of attacks. Our analysis has shown that in order to compromise this system, the attacker could easily use a simple HTML injection.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCSW'11, October 21, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-1004-8/11/10 ...$10.00.

[1]http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf

CONTRIBUTION. The contribution of this paper can be enumerated in the following main points:

1. Firstly, we propose to view the Cloud control interface security as an important and challenging research topic, additionally marked by its high impact factor for many stakeholders.

2. Secondly, we show that signature wrapping attacks remain a serious threat, as they are yet to be resolved or understood. We pair this with giving an overview of the (in)secure countermeasures.

3. Thirdly, we devise a methodology of investigating "black box" cloud implementations by making claims as to how SOAP message verification works in the Amazon EC2 cloud.

4. Fourthly and lastly, we show that the pure browser-based solutions do pose other, equally unresolvable problems through mounting different XSS attacks on the Amazon EC2 and S3 interfaces.

RESPONSIBLE DISCLOSURE. All the vulnerabilities found throughout our research have been reported to Amazon and Eucalyptus security teams. We have closely worked with both security teams and put forward the solutions for fixing the issues that have been identified. Subsequently, we monitored the countermeasures as they were being implemented.

RELATED WORK. Cloud security is an emerging research topic, already addressed in many academic and research-based publications. A good overview of cloud security issues is given by Molnar and Schechter who investigated advantages and disadvantages of storing and processing data by the public cloud provider with regards to security [23]. The authors detail the new kinds of technological, organizational, and jurisdictional threats resulting from the cloud usage, as they also provide a selection of countermeasures.

Ristenpart et al. analyzed the physical placement of new allocated virtual machines in Amazon EC2 [26]. They showed that an attacker can allocate new instances as long as one is placed on the same physical machine as his victim's instance. Afterwards, the attacker can exploit data from the victim's running instance using cross-VM side-channel attacks.

The attacker model given by Akhawe et al. [2] can be used to formally analyze the attacks in cloud computing scenarios. However, their initial approach is limited to HTTP communication only, and it does not take into account application layer messages like SOAP. In a similar scope, the formal modeling approach for Web Service security proposed by Bhargavan et al. [6] gives good advice on how to secure Web Service communication. However, applying their approach would not have fended the attacks described in this paper.

In 2009, Gruschka and Lo Iacono examined the security of the Amazon EC2 cloud's interfaces [16]. They showed how XML Signature wrapping attacks can be performed to attack Amazon's EC2 service. They presented a vulnerability that enabled an attacker to execute any operation on the cloud control, while being in possession of a signed control message from a legitimate user. Due to the timestamp included in the control message, their attack required an intercepted control message still being used within the validity period of five minutes.

The risk and impact of Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF) attacks have been discussed in detail by Johns in 2009 [20] and in his earlier publications [30]. XSS plays an important role in several attacks that we explicate, as it delivers the necessary information to deploy the attack payload without user interaction or brute forcing. This is especially relevant in blended attacks, which take on several steps to trigger and deliver exploit code and payload. In this paper, we build upon this initial work and further investigate the full potential of XML Signature wrapping and XSS attacks targeting the Amazon and Eucalyptus cloud control interfaces.

PAPER OUTLINE. *This paper is organized according to the structure delineated below. The following section introduces the Amazon and Eucalyptus cloud services and the XML Signature specification. Section 3 outlines the new attack techniques that have been discovered and proved to work for the SOAP-based interfaces of the Amazon EC2 cloud. Section 4 provides a similar analysis for the Eucalyptus cloud framework. Subsequently, Section 5 analyzes existing countermeasures, and shows why they are not sufficient to ward off these new attack techniques. Afterwards, we move on to offering countermeasures that are capable of successfully responding to the new attacks. As a follow-up, Sections 6 and 7 give yet another attack vector to exploit the existing vulnerabilities of the Amazon and Eucalyptus cloud Web front-ends. In Section 8 we take a closer look at the impact capabilities of the whole array of the attacks in question. The paper concludes with future research directives in Section 9.*

2. FOUNDATIONS

To introduce relevant areas of interest of this paper, the following subsections will review the main prerequisites.

2.1 Cloud Control

From a conceptual standpoint, cloud services need some form of cloud control which enables users to manage and configure the service, whilst also preserving access to the stored data. In IaaS-based clouds the control interface allows to, for example, instantiate machines, as well as to start, pause and stop them. Machine images can be created or modified, and the links to persistent storage devices must be configured. It is therefore quite undebatable that the security of a cloud service highly depends on robust and effective security mechanisms for the cloud control interfaces.

Technically, the cloud control interface can be realized either as a SOAP-based Web Service, or as a Web application (We acknowledge that there are other types of implementations which are not in scope for this paper.) If the control interface is SOAP-based, then WS-Security [24] can be applied to provide security services. For the authentication purposes, security tokens (mainly X.509 certificates) and XML Signature can be employed. A problem that generally arises is that the WS-Security standard is vulnerable to signature wrapping attacks [22], which consequently may invalidate this authentication mechanism.

If the control interface is a Web application, security relies on SSL/TLS combined with some client authentication mechanisms. Our results show that username/password based client authentication may be highly vulnerable to XSS attacks, thus other methods should take preference (e.g. TLS client certificates).

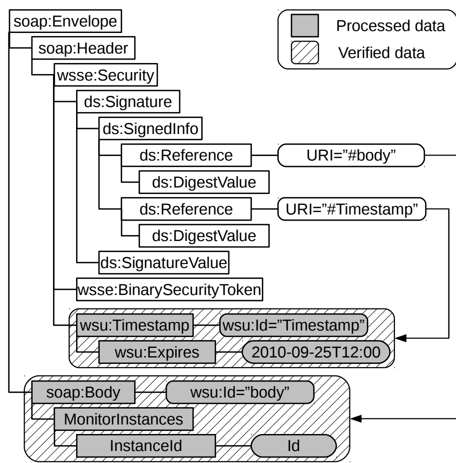

Figure 1: SOAP request sent to the EC2 interface

2.2 Amazon EC2 and S3 Control Interfaces

One of the most prominent cloud computing platforms is Amazon Web Services (AWS). It furnishes an array of products, e.g. computation services, content delivery, databases, messaging, payments, storage, and others, all made available to arbitrary companies and end-users. Elastic Compute Cloud (EC2) and Simple Storage Service (S3) remain the most popular among the chosen commodities. Amazon EC2 is a service that provides users with scalable computation capacity. Across a certain time period, the users can run their own virtual instances with customizable (virtual) hardware and operating system properties. Upon starting an instance using the EC2 cloud control, the user can for example access the instance over SSH (for Linux/Unix machines). Cryptographic keys for the SSH login may be similarly generated via the EC2 cloud control.

Amazon S3 gives its customers the possibility to store and access arbitrary data chunks (in the so-called *buckets*). Since EC2 does not provide persistent storage, it may be coupled with S3.

The two main interfaces are primarily responsible for EC2 and S3 services' control. The first one is a browser-based Web application (AWS Management Console). Logging in with their credentials, the user can check the status of the instances, run new instances, generate keys for communication with the running instances over SSH, create new buckets, or generate keys and certificates for controlling the cloud over SOAP- and REST-based Web Services. The Web application control interface is not intended for customers who own a huge number of machines that are dynamically started and stopped according to the computer power and storage needs. For this reason, AWS offers a complementary Web Services interface that gives the users a possibility to control their cloud over SOAP and REST-based services. Communication with these two interfaces can be automated.

The SOAP interface provides users with the same functionality as the AWS Management Console. The structure of SOAP messages, the names of the operations and their parameters are defined according to the XML Schema [12]. This schema is part of the WSDL document (Web Service Description Language [10]) that can be retrieved from the AWS Web site.

In order to provide integrity, authenticity, and freshness of the exchanged SOAP messages, the WS-Security standard is applied. This results in a message structure as depicted in Figure 1 (for reader's sake only the relevant parts are included). The `<soap:Envelope>`, `<soap:Header>`, and `<soap:Body>` elements delimit the structure of the SOAP message. The `<wsu:Timestamp>` element includes the message expiration date and therewith ensures its recentness. `<wsse:BinarySecurityToken>` [17] includes a Base64 encoded X.509 certificate that identifies the user. The `<ds:Signature>` element contains an XML Signature [4] authenticating the message issuer and protecting the integrity of the `<wsu:Timestamp>` and `<soap:Body>` elements. The `<MonitorInstances>` element indicates the (sample) operation to be called on the AWS interface.

The signature element and its content are created using the XML Signature standard. When verifying the integrity of the message, primarily the elements `<wsu:Timestamp>` and `<soap:Body>` are retrieved through the usage of the Id-based referencing. The values of the Id attributes are included as the parameters in the `<ds:Reference>` elements. Later on, the digest values over these elements are computed and compared to the values in the `<ds:DigestValue>` elements. Finally, the whole `<ds:SignedInfo>` element (including the two `<ds:DigestValue>` hash values) is normalized, a final hash value h is computed, and the signature from `<ds:SignatureValue>` is verified against h. In a case when all the checks are passed, the function defined in the SOAP body can be executed.

In addition to the EC2 SOAP interface described above, AWS provides three other types of Web Services interfaces: S3 SOAP Web Services interface with custom signature validation, AWS REST-based Web Services interface, and AWS XQuery Web Services interface. We are consciously deciding to exclude them from the discussion in this paper as they are not involved in the attacks we are covering.

2.3 Eucalyptus and Ubuntu Server Edition

While Amazon Web Services operates as a public cloud provider, the need for private cloud environments fostered the development of freely available open source implementations of the cloud systems. Among other advancements, the Eucalyptus cloud implementation [1] gained a lot of public attention and made its way into the well-known Ubuntu operating system (Ubuntu Server Edition). As of today, Eucalyptus is present within 25.000 installations of the world's most widely deployed software platform for Infrastructure-as-a-Service clouds.

As far as functionality is concerned, the cloud management interfaces of Eucalyptus were designed to copy the Amazon cloud control interface in order to support a switch from the prominent pre-existent Amazon EC2 cloud to an Eucalyptus cloud. Nevertheless, it must be stressed that the functionality and security mechanisms have been implemented independently. On that account, every Eucalyptus installation by default provides almost the exact same interfaces as the Amazon EC2 cloud. Furthermore, to make the message of our work clear, it has to be noted that the Eucalyptus SOAP interface provides the same methods as the Amazon EC2 interface described in the previous subsection. It also puts forth a customized Web front-end for a manual cloud administration.

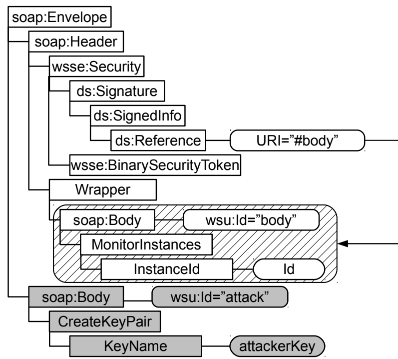

Figure 2: Classical Signature Wrapping Attack

2.4 XML Signature Wrapping

XML Signature [4] is the standard protection means for XML encoded messages, SOAP included. The so-called XML Signature Wrapping attack introduced in 2005 by McIntosh and Austel [22] illustrated that the naive use of XML Signature may result in signed XML documents remaining vulnerable to attacker's undetectable modifications. Thus, with a typical usage of XML Signature to protect SOAP messages, an adversary may be able to alter valid messages in order to gain unauthorized access to protected resources.

Generally speaking, the attack injects unauthorized data into a signed XML document alongside a possible restructuring in a way that the document's integrity is still verified, but the underlying consequence is that the undetected modifications are treated as authorized input during any further processing steps. In order to explain this attack, we assume that the attacker intercepts the SOAP message described in Figure 1 and needs to transform the operation in the SOAP body. The result of the signature wrapping attack is shown in Figure 2.

As shown in the figure, the original SOAP body element is moved to a newly added bogus wrapper element in the SOAP security header. Note that the moved body is still referenced by the signature using its identifier attribute `Id="body"`. The signature is still cryptographically valid, as the body element in question has not been modified (but simply relocated). Subsequently, in order to make the SOAP message XML schema compliant, the attacker changes the identifier of the cogently placed SOAP body (in this example he uses `Id="attack"`). The filling of the empty SOAP body with bogus content can now begin, as any of the operations defined by the attacker can be effectively executed due to the successful signature verification. In a given example, the adversary initiates a key generation process on behalf of the legitimate user being attacked.

3. AWS SOAP INTERFACE ATTACKS

Within the scope of a security analysis of Amazon's EC2 cloud control interfaces, we carried out an investigation of the SOAP message processing of the cloud control with respect to the applicability of XML Signature wrapping attacks.

3.1 Vulnerability Analysis

Authentication of a SOAP request message is done by checking an XML Signature that has to cover the timestamp header and the SOAP body. However, the overall structure of incoming SOAP messages—defined by the XML Schema [11]—is not checked at all. Therefore, it becomes possible to add, remove, duplicate, nest, or move arbitrary XML fragments within the SOAP request message—without the message's validity being affected.

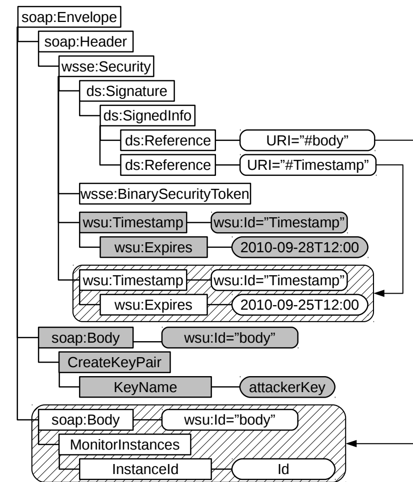

Figure 3: Signature wrapping attack type 1

We performed a set of SOAP requests that exploited this flexibility in SOAP message design. We have employed a validly signed SOAP message that triggers the operation `MonitorInstances`. This operation is used to gather status information on the user's EC2 virtual machine instances. Since the Amazon EC2 SOAP interface usually replies with quite meaningful SOAP fault messages in case of an error, we were able to easily test the Amazon EC2 SOAP interface for its signature wrapping resistance.

Remark: It is important to note that by using the signature wrapping technique we were able to invoke operations such as starting new VM instances, stopping any running instances, or creating new images and gateways in a victim's cloud environment—using the very same single eavesdropped SOAP request for the `MonitorInstances` operation (or any other operation of the EC2 SOAP interface).

SIGNATURE WRAPPING ATTACK VARIANT TYPE 1. The starting point for our security analysis was derived from the previous work done by Gruschka and Lo Iacono in 2009 [16]. Their attack used a forged SOAP request with a duplication of the signed SOAP body. Likewise, we duplicated the SOAP body of the `MonitorInstances` message, changing the operation in the first SOAP body to `CreateKeyPair`. We sent the forged message to the EC2 SOAP interface for verification. The message was successfully validated, and a new key pair for SSH access to an EC2 instance has been created. Conclusively, the EC2 SOAP interface validated the XML Signature only for the second SOAP body (which was not modified and hence verified successfully), but it used the first SOAP body for determining operation and parameter values. Supplementary tests with other operation names have indicated that an adversary could use this technique to trigger arbitrary operations. Still, all attacks must be performed within the five minute time frame enforced by the timestamp.

A slight attack variant circumvents the timestamp verifi-

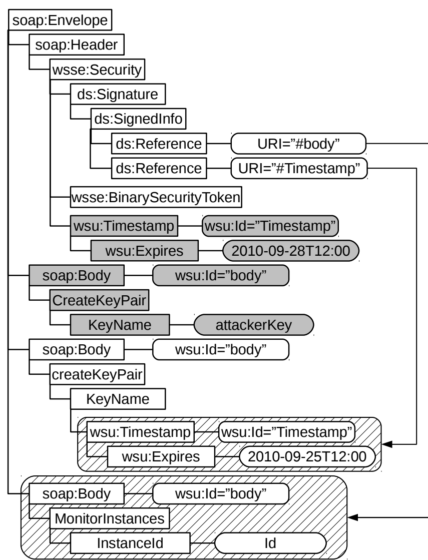

Figure 4: Signature wrapping attack type 2

cation, and therefore extends the attack to be independent of the time passing. Having duplicated the `<wsu:Timestamp>` element in the security header—the same approach used for the SOAP body before—we observed a similar behavior of the verification component: the first timestamp was compared to the current time, the second timestamp was verified for integrity. To sum up, this attack variant (shown in Figure 3) can be performed using arbitrary signed SOAP messages, even when their timestamp has already expired. The variant described above clearly breaks the timing constraints mechanism used in the EC2 SOAP interface, proving its potential for being used for execution of arbitrary operation invocation.

It is important to mention that the Id attributes of both *wrapped* and *executed* elements needed to be identical, as otherwise the message had been rejected.

Signature Wrapping Attack Variant Type 2. After reporting the first variant to the Amazon AWS security team, we were informed about a provision of a fix that disallowed duplications of the timestamp element. From this point forward, all the SOAP messages with duplicated timestamps *in the SOAP message's security header* were refused. However, it was still possible to have several `<soap:Body>` elements with the same `ID` attribute value within one SOAP message. For this reason, we continued our analysis focusing on moving the signed timestamp element to other positions within the document tree.

Figure 4 illustrates the first adapted wrapping attack on the EC2 SOAP interface. As it was no longer possible to duplicate the timestamp within the security header, we created three different `<soap:Body>` elements, and moved the originally signed timestamp element into the second body. Sending this forged SOAP message to the EC2 SOAP interface revealed that this attack technique indeed worked. The timestamp in the second body and the whole third body were checked by the signature verification component. The timestamp in the security header was attested for expiration, and the first body was interpreted as to determine the operation and parameter value.

We have also exposed other attack variants. For example, it was possible to duplicate the full SOAP security header. The first header included the timestamp that would be validated for its recency, and the timestamp in the second security header was corroborated by the signature validation component. Again, the first `<soap:Body>` element was executed, and the last one was verified for integrity. When compared to the type 1 vulnerabilities, same prerequisites and the same impact characterized the type 2 class.

Signature Exclusion Bug. The prerequisite for the above described signature wrapping attacks is that an adversary manages to obtain (namely eavesdrop, copy from a log file, etc.) a SOAP message with a valid XML Signature. Although this seems like a rather small obstacle (see also Section 3.2), we have detected another vulnerability with even less prerequisites: In the absence of an XML Signature, the signature verification component did not monitor any XML Signature at all, but nevertheless treated the message as validly signed. The task of user identification and authorization took place in other components relying solely on the X.509 certificate data from the `<wsse:BinarySecurityToken>` element—which can be present even if there is no signature. Hence, that SOAP request message was authorized to trigger operations on behalf of the owner of the X.509 certificate. To conclude, while performing an arbitrary SOAP request for any of the EC2 SOAP interface operations, an adversary needs only the public X.509 certificate of the victim. Since X.509 certificates are by definition considered to constitute public data, harvesting them from the Internet is not a major challenge for an adversary. Moreover, in Section 6.1 we discuss a download link XSS vulnerability that allowed us to gather valid certificates.

3.2 Attack Prerequisites

Based on the attack techniques highlighted so far, we continued our security analysis of the EC2 cloud control SOAP interface with surveying a degree of difficulty it takes for an adversary to get to the point where he can perform a successful signature wrapping attack.

Knowledge of a single validly signed SOAP request message remains the only prerequisite for a signature wrapping attack. Gathering such a SOAP message turned out to be quite an easy endeavor: many AWS developers seeking assistance post their SOAP requests on the AWS forums, which turned out to be a convenient source for signed SOAP messages. During the first attempt, we immediately recovered about 20 SOAP requests from multiple users of the `solutions.amazonwebservices.com` and `developer.amazonwebservices.com`. A slightly more sophisticated search would have very likely supplied us with even more results.

Remark: It must be stressed that SSL/TLS alone cannot solve the problem of signature wrapping attacks, because there are other ways to retrieve signed SOAP messages besides network tracking.

3.3 Analysis of the AWS Security Framework

Based on the attack findings described above, we performed an extensive security analysis of the Amazon EC2 cloud control SOAP Interface. By sending SOAP messages with different types of errors for different processing components of the AWS framework, we tried to determine the general architecture that Amazon uses for its SOAP interface services. Relying on publicly known best practices, we

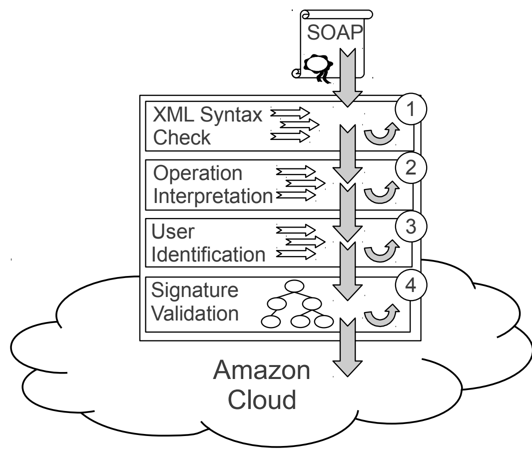

Figure 5: Amazon EC2 SOAP message processing architecture

SOAP-ENV:Envelope

```
<SOAP-ENV:Envelope
   xmlns:S[illegible]ENV="http://www.w3.org/2003/05/..."
   xmlns:aws="[illegible]lt/...">
   <SOAP-ENV[illegible]
      <SOA[illegible]
         <SO[illegible]
            <SOAP-E[illegible]
               SOAP-ENV:Sender
            </SOAP-ENV:Value>
            <SOAP-ENV:Subcode>
               <SOAP-ENV:Value>
                  aws:InvalidSOAPRequest
               </SOAP-ENV:Value>
            </SOAP-ENV:Subcode>
         </SOAP-ENV:Code>
         <SOAP-ENV:Reason>
            <SOAP-ENV:Text xml:lang="en-US">
               Invalid SOAP request. Could not parse XML
            </SOAP-ENV:Text>
         </SOAP-ENV:Reason>
         ...
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

soapenv:Envelope

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/..."
   xmlns:aws="http://webservices.amazon.com/AWSFault/...">
   <soapenv:Body>
      <soapenv:Fault>
         <faultcode>aws:Client.InvalidSecurity</faultcode>
         <faultstring>Request has expired</faultstring>
         <detail>
            <aws:RequestId>
               83264d5a-699d-48c3-83c1-c7eed8a38023
            </aws:RequestId>
         </detail>
      </soapenv:Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 6: SOAP fault messages for a SOAP request with a syntactical (left) and semantic fault (right)

assumed the Amazon Web Service interface consisted of a set of modules that perform specific tasks for every SOAP message received at the service interface. The order of these modules, and the amount of verifications performed therein usually is an important parameter of whether and how a typical web-service-specific attacks can be accomplished. Our goal was to gain as much information on this internal topology as possible, for a full view on the EC2 SOAP interface implementation.

Through sending hand-crafted SOAP messages to the EC2 interface, we effectuated a series of the SOAP-based tests. Each of these SOAP messages was carrying a different type of fault, causing the SOAP server implementation to raise diverse errors and respond with different types of SOAP fault messages. For instance, upon processing a SOAP message that contained a basic syntactical fault in the SOAP message's XML structure (e.g. a missing `'>'` character in the XML syntax) we received a SOAP fault message with a general XML structure as illustrated in Figure 6 (left). Please note the way the XML tag names are equipped with prefixes (e.g. `"SOAP-ENV"`). Though usually there is no semantic relevance for the choice of these namespace prefixes, they nevertheless tend to change for different XML frameworks, hence allowing a differentiation on a SOAP fault message's origin.

A second test was performed with the use of SOAP message with correct XML syntax but faults on the semantic level. As a result, the EC2 SOAP interface responded with a SOAP fault message as well, but this time there was a remarkable difference in the way the XML data was serialized. Figure 6 (right) shows an example of such a SOAP fault, received in reply to a SOAP request with an expired timestamp. Note the differences in how the XML namespaces are chosen (here: `"soapenv"`). Hence, it is reasonable to assume that both SOAP fault messages have been generated by different SOAP frameworks.

Similarly, test SOAP messages containing other types of faults, such as data type violations in operation parameters, invalid XML Signatures, or X.509 certificates have been used, as they were not known to the Amazon EC2 user database. We also performed tests with SOAP messages that contained two or more of these faults at the same time in order to see which fault the EC2 SOAP interface complained about first. This way, we have managed to identify the order in which the particular tasks are performed, the ways in which they accessed the XML data from the SOAP messages, and the estimated modularization architecture used within the EC2 SOAP interface.

The results of this analysis are depicted in Figure 5. As can be seen, the AWS SOAP interface processes the incoming SOAP messages in (at least) four separate logical steps, implemented by separate modules.

XML SYNTAX CHECK: In a first step, the XML parser performs a simple XML syntax check (so-called *well-formedness*). If even a single one of the XML tags is not properly closed or a namespace declaration is missing, the interface returns a SOAP fault. This step is most probably done by an independent XML parser, as the namespaces and the XML structure in the SOAP responses differed from the SOAP responses that were returned after processing of well-formed SOAP requests (see above).

OPERATION INTERPRETATION AND TIME CONSTRAINTS: In a second step, the XML processor reads and interprets the content of the SOAP request. First, it validates the time given within the `<wsu:Timestamp>` element. Then, it reads the `<soap:Body>` element, validating the contained operation name (e.g. `MonitorInstances`) and the number of its parameters. In all probability, this is obtained by using a *streaming* XML parser (such as SAX or StAX), since on duplication of the `<wsu:Timestamp>` or `<soap:Body>` elements only the first occurrence of that element is interpreted. This can be deemed as typical behavior for implementations that use streaming-based XML processing approaches, since these tend to interrupt message parsing immediately after having processed the first occurrence of the particularly interesting XML element. (Remark: This simple syntax check does not detect changes to the structure of the SOAP document, thus our attack messages are passing this step without any issues).

As can be seen by all the signature wrapping variants, the `wsu:Id` attributes of the wrapped and executed elements have to stay *equal*. Therefore, we assume that the Ids of processed elements are extracted and passed to the further XML Signature verification step.

USER IDENTIFICATION AND AUTHORIZATION: A third step attempts to identify the user by processing the X.509 certificate contained in the `<wsse:BinarySecurityToken>` element. The certificate determines the customer account of the Amazon user, thus performing solely the SOAP request's authorization task (and leaving *not* the authentication out).

XML SIGNATURE VERIFICATION: The last step before the operation in the SOAP message is executed, comprises of

XML Signature verification. The `URI` attributes of the XML Signature are dereferenced, i.e. the XML processor searches for XML elements that contain a `wsu:Id` attribute with the same identifier string value as indicated in the `URI` attribute of the `<ds:Reference>` element. Hence, for regular SOAP requests, this search returns the `<wsu:Timestamp>` and `<soap:Body>` elements as determined within the step two component. Then, hash value calculation and signature verification is performed for those elements. If this task fails, the SOAP message gets rejected, otherwise the operation determined in the step two component is performed on the Amazon EC2 cloud system.

In addition to accommodating verification of signature and digest values, this step checks if the elements being validated include the same `wsu:Id` attributes as the elements being processed in step 2. This grants the approval for the communication between the modules for *Operation interpretation* and *Signature validation*, which were there to attempt prevention of the signature wrapping attacks. However, allowing for multiple equal `wsu:Id` attributes in the SOAP message has opened possibilities for new variants of signature wrappings.

For the XML processing model of the last step we suppose that the `URI` dereferencing and determination of the signed elements is embedded in a tree-based XML Parser. This is due to the observation that tree-based XML parsers tend to keep an internal mapping of `wsu:Id` values to tree nodes, which is updated every time a new `wsu:Id` is found in the XML parsing process. Thus, if a `wsu:Id` value occurs twice within the same XML document, this mapping is overwritten and effectively points to the last occurrence of that `wsu:Id` value only. This behavior can e.g. be seen with the common Oracle (formerly Sun Microsystems) implementation of the Java XML Digital Signature API [28].

3.4 Attack Rationale and Assessment

The core misconception that enables attack techniques of signature wrapping and alike, lies in the separation of task modules within SOAP processing frameworks. As a result of this separation, different modules access the same XML document in a different way. Moreover, dissimilar modules may even use different computing paradigms, e.g. DOM based and streaming based SAX/StAX XML processing.

In the most common case, this deviation exists between the XML Signature verification module and the application logic implementation. The XML Signature verification typically locates the `<ds:Signature>` element at a certain position within the XML document (for SOAP messages, this is the `<wsse:Security>` header element), which then uses the contained `URI` references to search for the signed XML contents. In contrast, the application logic usually employs a different access approach, e.g. searching for the `<soap:Body>` element occurrence anywhere within the XML document. Subsequently, this deviation between the accessing mode of signature verification and application logic causes the vulnerabilities exploited by signature wrapping attacks.

For the attack variants here-presented, the first two attack techniques show typical instantiations of this deviation issue. To fend the former attack type, Amazon enforced *some* restrictions on where a signature-referenced XML element may be placed within the document. However, the latter attack techniques (Fig. 4) immediately bypassed these restrictions. This is due to the fact that the restrictions did not eliminate *all* deviations that could occur between signature verification module and application logic. These attack techniques prove that signature wrapping attacks are not well-understood and their complete elimination is complicated.

Interestingly, the final attack technique detailed above also can be seen as a variant of a signature wrapping attack. By omitting the XML Signature completely, with the exception of the required `<wsse:BinarySecurityToken>` element, the AWS framework legitimated the SOAP request for the user identified by the X.509 certificate contained in that element. Having taken a closer look at the AWS framework architecture (cf. Figure 5), we could have indicated that user authorization and signature verification (i.e. authentication) have been separated into distinct modules as well. Hence, the user authorization module can be seen as a particular kind of application logic that performs the sole task of determining and authorizing the user. In contrast, in this case it is not the XML document access method that is exploited for its deviation, but it is the deviation in assumptions that both modules make. The assumption behind the user authorization component is that there *exists* an XML Signature that enforces both message integrity and user authentication.

On the other hand, the assumption behind the XML Signature verification module is that *every* XML Signature contained in the SOAP message must be verified successfully in order to allow the SOAP request to pass, thus providing user authentication. Clearly, it does *not* enforce the existence of any such XML Signatures. This deviation in assumptions is what lead to this kind of vulnerability and exploit technique. Though being rather easy to fix, this attack technique nevertheless demonstrates a fundamental flaw in the typical separation-of-duties approach within the common Web Services frameworks.

To summarize what has been learned thus far, the attacks found in the Amazon EC2 cloud control SOAP interface are just scratching the surface of what is likely to be present in many of today's Web Service applications: the separation of tasks into distinct modules may easily lead to interoperability issues that can be in turn exploited for real-world attack techniques.

4. EUCALYPTUS SOAP INTERFACE ATTACKS

To analyze the Cloud control interface of Eucalyptus, we used a default cloud installation of the Ubuntu Server Edition, which provides an extended version of the original Eucalyptus framework [1].

4.1 Vulnerability Analysis

During our investigation, we have determined that signature wrapping attack techniques can be successfully applied to Eucalyptus. However, the techniques applied in the Amazon case were not functional, since Eucalyptus detects multiple identical Id attribute values, and rejects such SOAP messages. More precisely, in our analysis we discovered that an attacker can use a slightly modified *classical* wrapping attack technique to execute an arbitrary function without a time limitation. We give an example of a SOAP message of that sort in Figure 7.

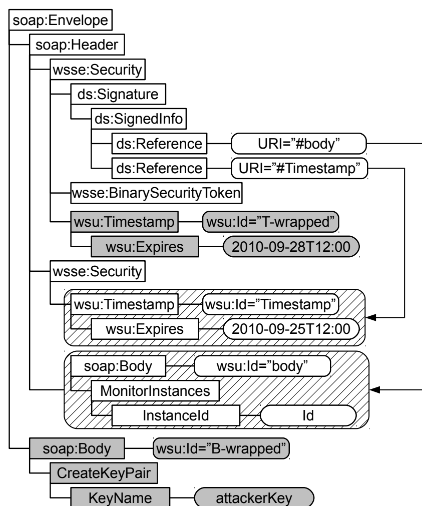

Figure 7: Successful signature wrapping attack on the Eucalyptus SOAP interface

As the Eucalyptus SOAP interface validates the format of incoming SOAP messages against an XML schema, the attacker cannot duplicate the SOAP body element or copy the signed elements directly to the SOAP header. For the attack to be feasibly executed, signed elements have to be copied to a newly created deeper-nested elements. For this purpose, we have chosen a duplicated security header element that does not violate the SOAP message XML schema. Through this process, the attacker can move the signed body and the timestamp elements to this newly allocated place.

Remark: This should be seen as a proof that Schema validation alone does not protect against signature wrapping attacks.

In addition to the SOAP message structure, the Eucalyptus validation framework checks for duplicated `wsu:Id` attribute values in the XML document. Conversely, it does not check if the processed data items have the same `wsu:Id` values as the signed data. Therefore, it was possible to use different `wsu:Id` attributes for the *executed* body and timestamp elements, which then had a potential to convey arbitrary content.

4.2 Attack Prerequisites

To execute an attack on Eucalyptus, an adversary must be in possession of a single validly signed SOAP message of the victim. It must be stressed once again that SSL does not prevent such attacks, since the SOAP messages in question can be retrieved in many different ways besides the network sniffing.

4.3 Analysis of the Eucalyptus Security Framework

Eucalyptus Framework is an open source private cloud provider. Therefore, there was no need for an extensive "black box" analysis. After analyzing the source code we found out that Eucalyptus uses for XML Security processing Apache Rampart – the security module of a widely used Apache Axis2 Web Services Framework [29]. Further tests of the Rampart module using various deployment properties have approved its vulnerability to signature wrapping attacks.

4.4 Attack Rationale

The problem in utilizing fixes of this vulnerability lies in the fact that Eucalyptus is deployed on various and numerous privately hosted servers. Therefore, each Eucalyptus administrator has to manually update his server version. Assuming a large number of installations (according to Eucalyptus there are more than 25.000 customers), we are doubtful that this attack will be mended on each server within a short period of time. This is arguably one of the largest downsides of relying on a private cloud infrastructure. In comparison to Eucalyptus, AWS developers could patch up the attacks and afterwards directly deploy fixes to all the running services. The fact that the vulnerability could be found on one of the leading Web Services frameworks (Rampart) pinpoints to the issue that it is not properly understood. Fixing this vulnerability on the Apache Rampart distribution is of an enormous importance, since it is deployed on a large number of business processing servers.

5. COUNTERMEASURES TO SIGNATURE WRAPPING ATTACKS

This section presents a number of countermeasures for signature wrapping and discusses their effectiveness in regards to the attacks presented above. Surprisingly, although (signature wrapping attacks are known since 2005 [22]), only few effectual countermeasures have been proposed in the literature, and even fewer have been implemented. This might be explicated by the difficulty of finding a formal model for this novel type of attack.

The first countermeasure against signature wrapping was elaborated on by McIntosh and Austel in 2005 [22]. They proposed to validate each message against an appropriate security policy. Still, most of the countermeasures were evaded by the authors themselves.

Similar requirements were furnished by Bhargavan, Fournet and Gordon [6, 7]. Their formal analysis of WS-Security [24] resulted in claims about the selection of items viewed as necessary parts for a security policy: The elements `<wsa:To>`, `<wsa:Action>`, `<soap:Body>` are mandatory to be present and signed. If present, the `<wsa:MessageID>` and `<wsu:Timestamp>` elements have to be signed as well. It is furthermore recommended to use X.509 certificates for authentication. Most of these items are covered by the EC2 SOAP interface requirements—with the exception of WS-Addressing, which is not supported by EC2. Failure of a formal analysis can be explicated quite simply: The model did not cover the semantic of the signature wrapping attacks. To provide such a semantic is a major research challenge, and a prerequisite for a formal analysis.

Often stated as another countermeasure, XML Schema validation can also help detecting SOAP message modifications used in a signature wrapping attack. However, current Web Service frameworks by default do not perform XML Schema validation, mainly due to the performance impacts of the validation process. Furthermore, even if present, XML Schema validation does not guarantee to fend signature wrapping attacks since XML schemas are *extensible*. (We have shown how to exploit this fact in the Eucalyptus attack message.)

For example, the SOAP 1.1 [8] specification—which is used by the EC2 SOAP interface—allows arbitrary elements inside the SOAP envelope after the body element. Thus, schema validation against this XML Schema would not be alarmed by any of the attacks presented in Section 3. On the other hand, given a hardened XML Schema that closely matches the intended SOAP message structure, XML Schema validation would have detected the additional bodies in the Amazon messages of signature wrapping attacks of type 1 and 2. A full analysis on the effectiveness of XML Schema validation in terms of fending signature wrapping is given in [19].

Another line of research can be summarized under the term "in-line approach", and was analyzed by Rahaman et al. [25] and Benameur et al. [5]. With this technique, additional information on the structure of a SOAP message is specified (and signed) in the header. However, due to the flexible structure of a SOAP message, these approaches can easily be circumvented, and some operational signature wrapping attacks in presence of an in-line approach countermeasure have been explored [13].

In [12], examples for an informal semantics for XML Signature were given. Nevertheless, a *full* semantics must be much more complex, as the namespace-based attacks on XML Signature have shown [18].

Another common countermeasure approach referred to as "see what is signed" is constituted by the fact that the application logic is only able to notice the XML content that was digitally signed, instead of attempting to parse and process the original XML message. This approach is not vulnerable to signature wrapping techniques (including the attacks presented in Section 3, since there is no way for the application logic to access ("see") non-signed data. A clear disadvantage of this procedural framework is that the interface between XML Signature verification module and application logic implementation is no longer appropriately particularized. This evokes to several issues (e.g. in presence of dedicated XML security gateways) that render this approach infeasible for many real-world applications.

In conclusion, the best countermeasure approach would be to enhance the interface between the signature verification function and the business logic. In this approach (see also [13]), the signature verification returns some sort of position description of the signed data, next to a Boolean value. The business logic may then decide if the data about to be processed has been signed or not.

6. AWS SCRIPT INJECTION ATTACKS

We have discovered two script injection vulnerabilities in the AWS management console web interface. The first vulnerability was difficult to exploit and targeted users of the Amazon AWS management interface only. The second vulnerability, found in the Amazon shop interface, made the attacks on the Amazon cloud possible, due to the login credentials being shared between the two systems.

6.1 Amazon Download Link Vulnerability

The first script injection vulnerability we discovered on the `aws.amazon.com` domain was caused by a download link used to retrieve X.509 certificates issued by Amazon. The purpose of our attack was to extract certificates of other users by exploiting this security bug. The vulnerability was rather hard to exploit, as in order to succeed, it required several preconditions to have been met. Nevertheless, it had the capacity to extract the public certificate content necessary for deploying some of the aforementioned attacks, and it was capable of sending relevant data to an arbitrary attacker-controlled domain. The following paragraphs will explain the attack and the steps guaranteeing the retrieval of the token data.

The server-side script, providing the X.509 certificate download link, accepted several GET parameters. Two of them were relevant for the attack, as they specified the name and the extension for the certificate, while another parameter outlined its *actual content* to download. This permitted a user to download a file with any desired name and content to their own browser. The possible attack scenario derived from the aforementioned conditions was the following: First, the attacker has to send a manipulated script link to the logged-in victim. By doing so, the attacker can force the script to generate an HTML file containing JavaScript code. This file then provokes a script injection attack taking place on the `aws.amazon.com` domain.

Two problems have emerged during the exploit code's testing: First, the server-side logic behind the script encoded a group of injection-critical characters such as `<,>` to HTML entities, thus rendering most attempts to generate HTML tags useless. To bypass this restriction, we made use of UTF-7 encoding [15] which for example represents the character `<` by the sequence `+ADw-`. The URL shown below contains the injection sequence <script>location=name</script> in UTF-7 encoding, demonstrating how the encoding routines could be bypassed.

```
https://aws-portal.amazon.com/gp/aws/developer
/account/index.html?ie=UTF8&filename=attack.html
&content=%2B%2Fv8%2BADw-script%2BAD4-location
%3dname%2BADw-%2Fscript%2BAD4-&action=download
```

The `location=name` assignment allows an attacker to execute arbitrary code stored in the DOM property `window.name`. Once set by *Domain A*—, this property will–resist any page refresh and even page changes to *Domain B*. The exception occurs when it is being overwritten or deleted during navigation. The attacker can specify the payload to be executed, by luring a victim onto a malicious page setting `window.name`, and then redirecting him to a page containing a JavaScript vector making use of `window.name`. The assignment to the magic `location` property ensures that the user agent location is actually changed to the given value. Thus, setting it to *javascript:eval(payload)* will execute the payload from the JavaScript URI, but not leave the `aws.amazon.com` domain context.

Internet Explorer is known to "sniff" for the proper character encoding to be used in case when no character set is given via meta info or HTTP header. This feature enabled the UTF-7 encoded exploit-trigger to execute without further modifications. At present, many recent browser versions were affected by this attack technique—note the UTF-7 Byte Order Mark used in the URL.

The second barrier preventing execution of the exploit code was the `content-disposition:attachment` header set by the affected script. We needed a way to display the content of the manipulated URL without triggering a file download dialog on the impacted browser. Again several Internet Explorer versions allowed us to do this by using a technique published by the Japanese security researcher Kanatoko [21]. The malicious URL had to be set as the `src` attribute for

an existing iframe with a short delay using the JavaScript function `setTimeout()`.

By combining all the mentioned techniques and prerequisites, an attacker could perform a script injection attack against logged-in victims. The script to download the certificate generated the payload to execute JavaScript via UTF-7 encoded HTML. The content-disposition headers' bypass trick then enabled the attacker to not only force the malicious code to be rendered and executed on the domain `aws.amazon.com`, but also to read that domain's HTML body. This of course included the section providing the certificate download, the authentication keys, and other sensitive data.

6.2 Amazon Public Stored XSS

Up till now, there were more reliable ways for an attacker to get hands on the necessary tokens to perform the aforementioned signature wrapping attacks. One of the biggest architectural flaw on `amazon.com` is the shared login session between the Amazon shop and the Amazon AWS management console interface. Once a user is logged into the Amazon shop, the login session for the Amazon AWS interface is also being created, despite the differing sub-domains `aws.amazon.com` and `www.amazon.com`. If a sophisticated attacker is behind the onset, a reflected or in a worse case - stored Cross Site Scripting (XSS) attack could cause harm and issues way beyond the theft of login credentials for the shop, or ordering items to an altered delivery address. We searched the Amazon shop for several kinds of XSS vulnerabilities and manged to expose a persistent XSS in the Amazon discussion forums, a frequently visited and public area, likely to attract many users and providing a lot of traction for attackers.

The attack we managed to perform is just as simple as it is effective. The attacker has to create a new discussion topic on either a shop item, a user-generated tag or other entities. Upon creation of the topic, the headline for the discussion topic will be reflected without proper encoding, thus allowing the injection of arbitrary HTML code. This has allowed us to include script tags or other active markup forcing the user agent to execute JavaScript on the `www.amazon.com` domain.

However, it is not possible to just inject arbitrary attack vectors, since Amazon uses a padding technique to convert incoming code into non-executing and broken markup to interfere with possible JavaScript execution. The JavaScript `<script>alert(document.cookie)</script>` for instance is transformed by the filter mechanism into something like the demonstration piece included below.

```
<script><span style=" font-size:0;=""></span>
alert(document.cookie<span style="
font-size:0;=""></span>)</script>
```

Still, this mechanism doesn't effectively keep an attacker from creating a functional attack vector but just delays the whole process. The attacker is forced to study the positions of the code padding and work-out a vector that is capable of reacting to this manipulation.

We nevertheless managed to create a persistent and public JavaScript injection and XSS attack against `www.amazon.com`. The code bypassing the padding protection is shown below. The JavaScript comments have been positioned exactly this way to defuse the padding sections, and leave the actual JavaScript payload working and ready to execute.

```
<!-- Input -->
<img onerror='/*123456789*/alert/*123456*/( cookie )' src=1>
<!-- Output -->
<img onerror="/*123456789 <span style=" font-size:0;
=""></span*/alert/*123456 <span style=" font-size:0
;=""></span>*/( cookie )" src=1>
```

The consequence of getting arbitrary JavaScript payload to execute is severe. An attacker can extract and steal the cookie data via `document.cookie` or alternatively try to lure the victim into leaking sensitive data by creating a forged login form. This kind of attack can be called in-site phishing, since a vulnerability in the phished site is used to harvest data with disastrous intent. Software and in-built mechanisms to protect a user from XSS attacks will not provide any shelter against this category of attacks due to their persistence – and not incapacity to be passed by via suspicious parameters.

6.3 Analysis of the Amazon Website Security Model

Another issue that we have pinpointed is that the Amazon Website as well as the AWS management console contain more security problems besides the ones already mentioned. None of the tested Amazon Websites utilized software to prevent the site from being loaded in a frame. An attacker can entice victims onto a malicious Website containing a frame pointing to the Amazon Website, which in turn may be overlapped by another frame and tunnel clicks or similar user interactions to the site overlapped. G. Rydstedt et al. [27] have drawn attention to the dangers of this so called 'click-jacking' technique, as they also pointed out efficient countermeasures and erroneous yet common frame buster implementations [3]. It must be stated that many of the critical forms used to setup user preferences, add one or more credit cards to the users payment portfolio as well as address changes, were not immune against CSRF attacks using a token or similar mechanism.

We believe that the precedence of Amazon AWS and the Amazon Shop sharing login sessions should cease. A vulnerability in the shop system automatically influences the AWS management console and vice versa. Additionally, XSS vulnerabilities in both systems can be used to extract cookie data, since Amazon avoids usage of `HTTPOnly` cookies [32]. These are furthermore shared between the SSL protected AWS management console and the usually HTTP-only driven store. In consequence, an attacker is able to easily eavesdrop on the victim in a man-in-the-middle attack and get hands on the session cookies for the AWS area without applying attempts to circumvent the protection delivered by the SSL [9].

7. EUCALYPTUS SCRIPT INJECTION ATTACKS

Our tests indicated that the cloud management web interface of the commonly used Eucalyptus software is equally vulnerable against Cross Site Scripting attacks. In-depth research has explicated that similarly to the aforementioned AWS attack vector a simple yet effective HTML injection can be used to fully compromise a cloud control web interface and remote control a logged in admin user. These attacks are not of academic interest, and therefore we do not analyze them in-depth.

It is recommended to apply protective measures to prevent hijacking and injection attacks against web-based cloud management interfaces that meet the requirements for highly critical web applications. A cloud control interface can serve arbitrary and subjective purposes – as long as browser and web application security are being left out and downscaled, all assets controlled by these interfaces are hard or even impossible to protect.

8. ATTACK IMPACTS

Exploiting any of the aforementioned vulnerabilities of the SOAP-based Amazon EC2 cloud control interface would enable an adversary to gain control over all cloud instances of the particular Amazon customer. Dependent on the type of services that a client operates via the Amazon EC2 cloud, the possibilities for getting malicious are endless.

The foremost obvious action an adversary may perform consists of creating and starting new virtual machine instances, which can then be put to use for one's own purposes. For instance, they can be exercised to send spam or phishing mails, for performing Denial of Service attacks, or for executing arbitrary calculations at the victim's costs (which will be charged to the adversary's cloud usage). It must be noted that all of these these attack scenarios could have been performed in other ways as well, e.g. by using a stolen credit card number or an intercepted authentication cookie.

What is more threatening, is the fact that the adversary gains complete and unlimited access to each and every single one of the victim's existing virtual machine images[2]. Multiple ways of exploiting this phenomenon can be brought about. For instance, the adversary is able to right away eavesdrop on all kinds of data that are contained within any of the existing virtual machine images. This may range from private keys used in SSH or HTTPS servers over business data and customer account lists up to information regarding the processes that run in the victim's service applications. Especially the latter poses a tremendous threat. The adversary may uncover the business secrets that are stored in the applications, which inevitably makes him even more able to *change* the way these applications work to his advantage.

8.1 Example Scenario: Attacking Twitter

An interesting attack scenario demonstrating the impact of the attacks we have shed light upon would involve a targeted attack and several parties to unfold the full impact. Let's assume an attacker that intends to distribute JavaScript-based malware on a global level. In this case, the possible attack would comprise of three steps, which will be discussed in this section. For this attack scenario, we chose two potential targets: Amazon S3 storage and the popular Twitter micro-blogging service of more than 140 million users [31].

ATTACKING AMAZON: The attack requires the presence of a Cross Site Scripting attack in either the Amazon AWS management console, the Amazon shop, or any other website sharing login credentials with the AWS management console. Ideally, the vulnerability results in a persistent Cross Site Scripting attack allowing the injected vector to easily bypass protective mechanisms like NoScript or the IE8 XSS filter. The attacker prepares a payload for the exploit capable of reading the victim's cookies or accessing username and password in plain text in case the victim uses the browser's password manager to store the Amazon login data. This attack technique is often being referred to as Logout XSS [33].

VICTIM SELECTION AND HARVESTING: The attacker needs to pick a victim – ideally a person employed by Twitter and supplied with access to their AWS management console account. As soon as all potential victims are chosen, the attacker must make them visit the infected website of the Amazon estate. If a victim has JavaScript enabled, the exploit code will trigger and execute the malicious payload. In case the attacker succeeds, he attains access to the victim's cookie data, the login data including password, or the certificate.

DATA MANIPULATION AND EXPLOIT SPREADING: If the attacker was to harvest victim's public certificate, he could easily execute the signature exclusion attacks on the SOAP EC2 interface, granting himself an ability to modify the existing or setup the new virtual machine images (AMI). Else, if the attacker used the harvested login data to get access to the Twitter AWS management console, he would have also achieved access to the Twitter Amazon S3 buckets storing static content being deployed on `twitter.com`. This includes the `base.bundle.js` file that is deployed with every request to the `twitter.com` index page for logged in users. Manipulating this single script file would have thus affected every user logging into Twitter via the website having JavaScript enabled.

Apart from Twitter, many other high traffic websites and popular web applications utilize the services provided by Amazon. The list includes Secondlife, SurveyMonkey, SAP, the New York Times' website, Reddit.com and Foursquare.

9. CONCLUSION

In this paper, we have presented the results of our security analysis of the Amazon and Eucalyptus cloud systems. We have revealed several highly critical vulnerabilities in the EC2's SOAP and Web interfaces. Those would allow an adversary to gain root access to arbitrary virtual machines and Web applications hosted in these clouds, as well as gather arbitrary files and data from the Amazon S3 cloud, and the arbitrary installations of Eucalyptus clouds. Besides the tremendous impact of the attacks themselves, the fact that all these vulnerabilities were uncovered within a very limited time frame, must be considered to be of particular importance.

It shows that the complexity of such systems creates a large seedbed of potential vulnerabilities. Hence, cloud control interfaces are very likely to become one of the most attractive targets for organized crime in the nearby future ahead. The most important threat pertains to every vulnerability we found as impacting not just a single server or company, but all of the associated cloud users at once. Additionally, Cross Site Scripting attacks against Web-based cloud control interfaces have severe repercussions for the overall cloud security. They can easily be leveraged to extract sensitive information. Victims logged into the Web interface or using the browser-based password manager to store the cloud control interface login credentials can be impersonated straightforwardly. They risk having their login

[2]Note that this access does not include running instances, yet it covers *all* instance images available within the victim's EC2 account.

data be extracted and sent to arbitrary domains with few lines of exploit code. If carried out well, precise attack can affect several millions of users. SSO-based Web platforms sharing their login credentials with the targeted cloud control interface drastically enlarge the risk and impact of the attacks we have highlighted.

Finally, we have managed to show a large number of countermeasures for the attacks we described. We intended to explain as to what extent they are able to fend the particular attack types. Undoubtedly, the most important lesson learned from our analysis is that managing and maintaining the security of a cloud control system and interface is one of the most critical challenges for cloud system providers worldwide.

Acknowledgement

We would like to thank the Amazon and Eucalyptus security staff for their cooperation, and wish to note that throughout the collaboration both teams effectuated an excellent, productive, and highly professional communication.

We would also like to thank Xiaofeng Lou for his contributions.

10. REFERENCES

[1] Eucalyptus. http://open.eucalyptus.com/.

[2] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J. C., and Song, D. Towards a formal foundation of web security. In *CSF* (2010), pp. 290–304.

[3] Balduzzi, M. New Insights Into Clickjacking. In *OWASP AppSec Research* (2010).

[4] Bartel, M., Boyer, J., Fox, B., LaMacchia, B., and Simon, E. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation* (2008). http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/.

[5] Benameur, A., Kadir, F. A., and Fenet, S. XML Rewriting Attacks: Existing Solutions and their Limitations. In *IADIS Applied Computing 2008* (Apr. 2008), IADIS Press.

[6] Bhargavan, K., Fournet, C., and Gordon, A. D. Verifying policy-based security for Web Services. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), ACM Press, pp. 268–277.

[7] Bhargavan, K., Fournet, C., Gordon, A. D., and O'Shea, G. An advisor for web services security policies. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM Press, pp. 1–9.

[8] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. SOAP 1.1. *W3C Note* (2000).

[9] Callegati, F., Cerroni, W., and Ramilli, M. IEEE Xplore - Man-in-the-Middle Attack to the HTTPS Protocol. *Security & Privacy, IEEE 7*, 1 (2009), 78–81.

[10] Chinnici, R., Weerawarana, S., Moreau, J.-J., and Ryman, A. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Tech. rep., OASIS, Mar. 2004.

[11] Fallside, D. C., and Walmsley, P. XML Schema Part 0: Primer Second Edition. *W3C Recommendation* (2004).

[12] Gajek, S., Jensen, M., Liao, L., and Schwenk, J. Analysis of signature wrapping attacks and countermeasures. In *ICWS* (2009), IEEE, pp. 575–582.

[13] Gajek, S., Liao, L., and Schwenk, J. Breaking and fixing the inline approach. In *SWS* (2007), pp. 37–43.

[14] Gens, F. IT Cloud Services User Survey, pt.2: Top Benefits & Challenges. *IDC eXchange* (2008).

[15] Goldsmith, D., and Davis, M. RFC 1642: UTF-7 – A Mail-Safe Transformation Format of Unicode, Jul. 1994.

[16] Gruschka, N., and Lo Iacono, L. Vulnerable Cloud: SOAP Security Revisited. In *Proceedings of the IEEE International Conference on Web Services* (2009), IEEE Computer Society, pp. 625–631.

[17] Hallam-Baker, P., Kaler, C., Monzillo, R., and Nadalin, A. Web Services Security X.509 Certificate Token Profile. W3C recommendation, W3C, Jun. 2007. http://www.w3.org/TR/2007/REC-wsdl20-20070626.

[18] Jensen, M., Liao, L., and Schwenk, J. The curse of namespaces in the domain of xml signature. In *SWS* (2009), E. Damiani, S. Proctor, and A. Singhal, Eds., ACM, pp. 29–36.

[19] Jensen, M., Meyer, C., Somorovsky, J., and Schwenk, J. On the effectiveness of xml schema validation for countering xml signature wrapping attacks. In *Proceedings of the First International Workshop on Securing Services on the Cloud* (2011).

[20] Johns, M. *Code Injection Vulnerabilities in Web Applications – Exemplified at Cross-site Scripting.* PhD thesis, University of Passau, Passau, 2009.

[21] Kanatoko. Bypassing Content-Disposition:Attachment on Internet Explorer, 2007.

[22] McIntosh, M., and Austel, P. XML Signature Element Wrapping attacks and Countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services* (New York, NY, USA, 2005), ACM Press, pp. 20–27.

[23] Molnar, D., and Schechter, S. Self hosting vs. cloud hosting: Accounting for the security impact of hosting in the cloud. In *Proceedings of the Ninth Workshop on the Economics of Information Security (WEIS)* (2010).

[24] Nadalin, A., Kaler, C., Monzillo, R., and Hallam-Baker, P. Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). *OASIS Standard Specification* (2006).

[25] Rahaman, M. A., and Schaad, A. Soap-based secure conversation and collaboration. In *ICWS* (2007), pp. 471–480.

[26] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), ACM, pp. 199–212.

[27] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites.

[28] Sun Microsystems. *XML Digital Signature API*, 2006.

[29] The Apache Software Foundation. Apache Axis2.

[30] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium (NDSS)* (2007).

[31] Williams, E. Twitter Blog: The Evolving Ecosystem, 2010.

[32] Zhou, Y., and Evans, D. Why aren't HTTP-only cookies more widely deployed? In *Workshop on Web 2.0 Security and Privacy (W2SP)* (May 2010).

[33] Zuchlinski, G. The Anatomy of Cross Site Scripting. *Hitchhiker's World 8* (2003).