A1 – Lack of Authentication

By default Web services have no channel, method or data authentication. This means that anonymous access is often possible to Web services, and since Web services are often used as integration technologies anonymous access is available to back end systems like mainframe, ERP, CRM, and other systems. Lack of authentication can lead to the spoofing of user or administrative accounts, replay attacks, and cause privacy violations.

ENVIRONMENTS AFFECTED

All web services frameworks that use interpreters or invoke other processes are vulnerable to injection attacks.

This includes any components of the framework that might use back-end interpreters. Remember that web services are frequently used to provide a Web gateway performing protocol translation and data transformation for legacy back ends. If the Web service gateway is bypassed, it is highly likely these back end systems were designed only for benign environments.

VULNERABILITY

Attacker can spoof identity of service requester, service provider, substitute and tamper with data, the recipient of the Web service message has no way to differentiate between an legitimate and a spoofed message. Recipient has no way to identify from where message originated. Attacker can send the same message many times - message replay. Attacker can delay a legitimate messages - can cause problems in time sensitive applications such as exchanges and auctions.

```
POST /axis2/services/getCustomer HTTP/1.1
Content-Type: text/xml; charset=UTF-8
SOAPAction: "urn:echo"
User-Agent: Axis2
Host: example.com
Transfer-Encoding: chunked


<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
        <ns1:echo xmlns:ns1="http://example.com">
        <param0>Hello world</param0></ns1:echo>
```

A Web service that accepts the above message as authentic has no way to verify who sent the message, this also weakens the authorization process.

VERIFYING SECURITY

The goal is to verify that the service properly authenticates requests and responses and properly protects identities and their associated credentials, tokens, claims and assertions based on policy.

Automated approaches: Vulnerability scanning tools may be able to identify web services that are open to anonymous requests and responses, but beyond this have a very difficult time detecting vulnerabilities in custom authentication and session management schemes. Static analysis tools may be able to detect some authentication and token flaws.

Manual approaches: Code review and testing, especially in combination, are quite effective at verifying that authentication, token usage, and ancillary functions are all implemented properly. An architecture and policy review is necessary to find the right mix of security mechanism in a web service.

PROTECTION

Web service authentication begins with a secure communication channel, and verifiable credentials, tokens, claims, and assertions. First ensure that SSL is the only option for all authenticated parts of the service communications (see A9 – Insecure Communications), next the SSL channel should be further protected with Transport Layer Security (TLS) and client/server certificates for mutual authentication on the communication channel, all credentials, tokens, claims, and assertions are sent in signed, hashed and encrypted form (see A8 – Insecure Cryptographic Usage),. Ensure that these credentials, tokens, claims and assertions are then verified by the method. For example, one common Web service vulnerability is where the service requester signs the message, and the service provider checks to see if the message is signed before granting the request, however, the provider must also check that it trusts the signer!

Preventing authentication takes careful planning. Among the most important considerations are:

- Use SSL for a encrypted communications channel to protect the message and authentication tokens
- Use TLS to mutually authenticate the service requester and service provider

- Limit or rid your code of custom authentication and token schemes, such as <username>joe</username> <password>hard2guess</password> type functionality, passing an over-privileged system user like "MQUSER" or home grown single-sign on functionality.
- Use a verifiable token such WS-Security with SAML, Kerberos or X.509 token to hash, sign, and encrypt the requests and responses.
- Use nonce in message and ensure the recipient checks nonce cache to prevent replay attempts. Ensure that this nonce is hashed and signed, see previous point on verifiable tokens
- Use timestamp, and created/expired timestamp data to prevent session replay and other timing attacks. Ensure that timestamp data is hashed and signed, see previous point on verifiable tokens

- Do not allow the login process to start from an unencrypted request or response. Always start the login process from an encrypted request or response with a fresh or new session token to prevent credential or token stealing attacks
- Consider regenerating a new token upon successful authentication or privilege level change.
- Do not rely upon spoofable credentials as the sole form of authentication, such as IP addresses or address range masks, DNS or reverse DNS lookups, referrer headers or similar
- Remember to authenticate both the request and response. So both the service requester and service provider have inflow and outflow authentication mechanisms

SAMPLES

http://capec.mitre.org/data/definitions/95.html

http://capec.mitre.org/data/definitions/57.html

http://capec.mitre.org/data/definitions/36.html

REFERENCES

CWE: CWE-287 (Authentication Issues), CWE-522 (Insufficiently Protected Credentials), CWE-311
(Reflection attack in an authentication protocol), others.

WASC Threat Classification:
http://www.webappsec.org/projects/threat/classes/insufficient_authentication.shtml
http://www.webappsec.org/projects/threat/classes/credential_session_prediction.shtml
http://www.webappsec.org/projects/threat/classes/session_fixation.shtml

OWASP Guide, http://www.owasp.org/index.php/Guide_to_Authentication

OWASP Code Review Guide,
http://www.owasp.org/index.php/Reviewing_Code_for_Authentication

OWASP Testing Guide, http://www.owasp.org/index.php/Testing_for_authentication

A2 – Lack of Authorization
Security solely by obscurity is not reliable, many web services are deployed when meet business requirements. Security requirements are often generated by security assessments that occur late in the development process, since authorization is linked to operations,

functionality and data, once the application is developed, its frequently too late to bolt authorization code on after the fact.

Second, many organizations say they are using an authorization strategy, like Role Based Access Control ,in Web services environments; this glosses over the important issue of how the authentication claims are processed, and evaluated to populate the subjects that make the authorization in the first place.

ENVIRONMENTS AFFECTED

All web service frameworks are vulnerable to failure to lack of authorization. The default mode for all web service frameworks is to allow anonymous access.

VULNERABILITY

For one example of authorization in Web services, consider this from "Mission:Messaging: WebSphere MQ, PCI DSS, and security standards" by T. Rob Wyatt (http://www.ibm.com/developerworks/websphere/techjournal/0806_mismes/0806_mismes.html)
:

> "If the default settings for WebSphere MQ are not updated after installation, the authentication of remote users is only a good faith effort and can easily be bypassed
> …
> For example, when using a gateway or hub in combination with channel security, the hub is authorized to all legitimate destinations in the network. Therefore, any message passing through the hub is also authorized to these destinations -- even though it might not have originated from an authorized application or user."

This is typical of many Web services environments which are built to facilitate integration, and not to enable *security policy-based* integration.

To find this vulnerability, attackers enumerate endpoint hosts guessing service directories, paths, and methods to find unprotected services. The sheer complexity of the many small parts in a loosely coupled service means its likely that even if say WSDL is protected, the URL, XSD, registry or other component part of the service may broadcast anonymous attack surface.

Some common examples of these flaws include:

* "Hidden" or "special" Services, rendered only to administrators or privileged users that are accessible to all users if they know it exists. This is particularly prevalent with enterprise messaging systems that connect front end web applications to backend transactional legacy backbones.

* Applications often allow access to "hidden" files, such as static XML, XSD, WSDL or system generated reports, trusting security through obscurity to hide them.

* Code that enforces an access control policy but is out of date or insufficient.

* Services may be generated and deployed without developers and administrators knowledge. Many IDEs like Visual Studio and Eclipse can automatically generate service interfaces complete with WSDL and XSD, and these may end up being deployed onto a production system without understanding the side effects.

* Code that evaluates privileges on the service requester but not on the service provider. Conversely, code that evaluates privileges on the service provider but not on the response to the service requester leaves the requester vulnerable.

* Code that checks for a signature, but does not check if the signer is trusted

```
<SOAP:Header>
        <WSSE:Security>
                <ds:Signature>valid signature
                <dsig:SignatureValue>untrusted signer
                        <ds:Reference URI='#body'>
```

VERIFYING SECURITY
The goal is to verify that authorization is defined and enforced in a way that's consistent with policy.

Automated approaches: Both vulnerability scanners and static analysis tools have difficulty with verifying authorization, but for different reasons. Vulnerability scanners have difficulty in determining what the policy is that governs service methods, schemas, and metadata and determining which should be allowed for each user, while static analysis engines struggle to identify custom access controls in the code and link the service interface with the business logic. However, some static analysis tools may be able to detect use of standardized security mechanisms such as WS-Security and XML Signature.

Manual approaches: The most efficient and accurate approach for custom coded access controls is to use a combination of code review and security testing to verify the access control mechanism. If the mechanism is centralized, the verification can be quite efficient. If the mechanism is distributed across an entire codebase, verification can be more time-consuming. If the mechanism is enforced externally, the configuration must be examined and tested.

PROTECTION
Taking the time to define a policy, plan authorization by creating a matrix to map the subjects (like user accounts, groups, and roles) and the objects (like methods, and data) is a design time task that is difficult to bolt on after the fact. Throwing technology or even security standards like WS-Security, Kerberos, or others will not help much if the application is designed without an access control chokepoint or policy enforcement point in the first place. Its up to the designer to find all the locales where Web services must enforce access control on every service, policy, data, metadata and business function.

It is not sufficient to put access control into the initial service request and leave the business logic or data unprotected. It is also not sufficient to check once during the process to ensure the user is authorized, and then not check again on subsequent steps. Otherwise, an attacker can simply skip the step where authorization is checked, and forge the parameter values necessary to continue on at the next step. Further, its not sufficient for an access control to reside only on one part, both the service requester and service provider need to provision inflow and outflow security. Enabling service access control takes some careful planning. Among the most important considerations are:

* Ensure the access control matrix is part of the business, architecture, and design of the application. Ensure the tokens, claims, and assertions in the request and response messages are mapped into access control decisions

* Ensure that all services, policies, data, metadata and business functions are protected by an effective access control mechanism that verifies the user’s role and entitlements prior to any processing taking place. Make sure this is done during every step of the way, not just once towards the beginning of any multi-step process

* Perform a penetration test prior to deployment or code delivery to ensure that the application cannot be misused by a motivated skilled attacker

* Pay close attention to policy and metadata files. They should verify that they are not being directly accessed, e.g. by checking for a constant that can only be created by the library’s caller

* Do not assume that users will be unaware of special or hidden URLs or APIs. Always ensure that administrative and high privilege actions are protected

* Block access to all file types that your application should never serve. Ideally, this filter would follow the "accept known good" approach and only allow file types that you intend to serveThis would then block any attempts to access log files, xml files, etc. that you never intend to serve directly.

* Keep up to date with virus protection and patches to components such as XML processors, word processors, image processors, etc., which handle user supplied data

SAMPLES

http://capec.mitre.org/data/definitions/1.html
http://capec.mitre.org/data/definitions/58.html
http://capec.mitre.org/data/definitions/71.html
http://capec.mitre.org/data/definitions/36.html
http://capec.mitre.org/data/definitions/95.html

REFERENCES

CWE: CWE-325 (Direct Request), CWE-288 (Authentication Bypass by Alternate Path), CWE-285 (Missing or Inconsistent Access Control)

WASC Threat Classification: http://www.webappsec.org/projects/threat/classes/predictable_resource_location.shtml

OWASP, http://www.owasp.org/index.php/Forced_browsing

OWASP Guide, http://www.owasp.org/index.php/Guide_to_Authorization

A3 – Lack of audit logging

Audit logging systems should record critical systems events such as authentication, authorization, failures, and exceptions. By default web services record none of these. Lack of centralized or reasonable logging, particularly between services in a flow (the fiefdom problem - if there are three different ops groups or more... it can be hard to get the messages from one flow to go to one management area

ENVIRONMENTS AFFECTED
All web service frameworks are vulnerable to failure to lack of audit logging. The default mode for all web service frameworks is to not record audit log events.

VULNERABILITY
Web services do not have a native way to correlate requests and responses (hence loose coupling), however it makes reconstituting an end to end view challenging

Web services lack a common record format and publishing protocol for conveying security information.

VERIFYING SECURITY
The goal is to verify that the audit log captures audit events and provides a way to browse for the auditor to review what happened on the system.

Automated approaches: tools may check for the presence of a logging subsystem but lack an end to end method to ensure the events are captured and catalogued at runtime.

Manual approaches: Security testing can be performed to verify the audit logging mechanism by generating events and verifying that they are recorded properly. Both the audit log generation and audit log repository should be reviewed.

PROTECTION

Web services should generate an event log that is published to a secure environment. The audit log should record events that are useful to determine security critical events, identify and recover from errors such as those suggested by Chuvakin (see References):

- AAA (Authentication, Authorization, Access)
  - Authentication/authorization decisions
  - System access, data access
- Change
  - System/application changes (especially privilege changes)
  - Data change (creation and destruction are changes too)
- "Badness"/ Threats
  - Invalid input (schema validation failures)
  - Resource Issues
- Resource exhausted, capacity exceeded, etc
  - Limit reached (message throughput, message replays)
  - Mixed Availability Issues
- Startups and shutdowns
  - Faults and errors
  - Backups success / failure

The application's policy should determine if the application is able to function when the audit logging subsystem is unavailable.

SAMPLES

XDAS – Distributed Audit Services
http://www.opengroup.org/security/das/xdas_int.htm

http://capec.mitre.org/data/definitions/93.html

REFERENCES

"Design for Failure – A Conversation with Bruce Lindsey" ACM Queue, http://queue.acm.org/detail.cfm?id=1036486

“Eventually Consistent” by Werner Vogels, http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

“Logging – the Good, the Bad and the Ugly” by Anton Chuvakin, http://www.slideshare.net/anton_chuvakin/application-logging-good-bad-ugly-beautiful-presentation

“Logging in a Web services world” by Anton Chuvakin and Gunnar Peterson, IEEE Security & Privacy Journal

A4 – Lack of Security Policy
In many systems its difficult to assess whether the system is secure or not, because there is a lack of security policy. Web services are used in many different types of applications from Mainframes to Smart Cards so its up to the policy to define the expected security mechanisms.

ENVIRONMENTS AFFECTED
All web service frameworks are vulnerable to failure to lack of security policy. The default security policy for web service frameworks is to not to enforce access control on the web service communication channels, methods or data.

VULNERABILITY
Many web services security architectures are just a collection of ad hoc design and implementations rather than mechanism that reflect security policy decisions. Many web services apply controls in a haphazard manner but do not specify policy to cover the scope of Web services including operations, messages, methods, and communications channels

Robert Morris, Sr. once said: "Systems built without requirements cannot fail; they merely offer surprises - usually unpleasant!" The same is true for security services that lack policy statements.

VERIFYING SECURITY
The goal is to verify that the Web services security policy defines a security policy for the Web services communication protocols, application methods and data.

Automated approaches: some static analysis tools may check for the presence of a policy and whether the policy is attached to the web service.

Manual approaches: Security testing can assess whether there is an explicit, declarative policy, policy enforcement point(s) attached to the service, and whether these may be evaded.

PROTECTION
“Security should depend on policy not topology.”-Bill Gates Feb. 6, 2007

Declarative security policy should be defined for all Web services initiation, inflow and outflow message exchange patterns. The policies should mandate allowable and not allowable, for example:

- Communication protocols (e.g. HTTP, JMS)
- Message exchanges
- Methods
- Security tokens (e.g. SAML, X.509)
- Data integrity
- Replay protection (if a nonce is required)
- Timestamp usage
- Data encryption
- Key management

WS-SecurityPolicy provides one way to exert granular control over security policy at the transport (non-message level), message level security, and allowable crypto and token types.

Example WS-SecurityPolicy statement for requiring HTTPS:

```
<wsp:Policy wsu:Id="UTOverTransport" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
        <wsp:ExactlyOne>
          <wsp:All>
        <sp:TransportBinding
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
                  <wsp:Policy>
                        <sp:TransportToken>
                          <wsp:Policy>
        <sp:HttpsToken RequireClientCertificate="false"/>
                          </wsp:Policy>
                        </sp:TransportToken>
```

Example WS-SecurityPolicy statement for requiring HTTPS with a X.509 token for mutual authentication:

```
<sp:AsymmetricBinding
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
        <sp:InitiatorToken>
        <wsp:Policy>
        <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/Include
Token/AlwaysToRecipient">
                                                            <wsp:Policy>
```

```
          <sp:WssX509V3Token10/>
                                                    </wsp:Policy>
          </sp:X509Token>
…

…<sp:RecipientToken>
<wsp:Policy>
<sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/Include
Token/Never">
<wsp:Policy>
<sp:WssX509V3Token10/>
</wsp:Policy>
…
</sp:RecipientToken>
<sp:AlgorithmSuite>
<wsp:Policy>
          <sp:TripleDesRsa15/>
</wsp:Policy>
```

The assertions in the policy tell the assessor what security is to be used for the communications channel, and then the assessor can verify that these are in fact implemented at runtime.

Policy statement like WS-SecurityPolicy may be used to enforce policy decisions and as such these files and assertions become part of the access control architecture and require a high level of protection - through digital signature and verification.

SAMPLES

REFERENCES

“We Need Assurance!” Brian Snow http://www.acsa-admin.org/2005/papers/Snow.pdf

A5- BROKEN XML

The unifying construct in services is messages. The integration between the service requester and service provider is not binary integration (as with J2EE, DCOM), rather it is message document based, often XML. XML presents a different type and value system for the message that is then interpreted and mapped to the runtime language, e.g. Java or C#. There are two main ways XML creates vulnerabilities - structural representation through inconsistent types, values, and encoding; and behaviors such as parsing and validation

ENVIRONMENTS AFFECTED
All web service frameworks are vulnerable to attacks on broken XML.

VULNERABILITY

XML vulnerabilities apply to service requesters, service providers, and any intermediaries such as registries and Enterprise Service Buses. Broken XML vulnerabilities include:

– Not validating against a schema at all
– Using a DTD schema which may lead to XML Denial of Service (XDoS)
– Using a weak or lax schemas
• No using <max-length>…</max-length>
• Not limiting character set <xs:element name="charset" minOccurs="0">
• Not encoding all output <?xml version="1.0" encoding="utf-8" ?>
• Allowing type any and lax enforcement - <xs:attribute name="requestDate" type="xs:any"/>
• The standard web services security tokens like SAML and WS-Security tokens are in fact implemented in XML so the security tokens themselves are an attack vector, effectively they are vectors for anonymous attacks, because attacks targeting the parsing, canonicalization and transformation routines are inserted into security tokens are potentially executed before the token is authenticated.[BH]

VERIFYING SECURITY

The goal is to verify that the application has hardened it schema, that validation is occurring against the hardened schema wherever the XML is sued and that the parsing routines have been configured securely.

PROTECTION

– Ensure any SAX parsers are resilient to known Entity Resolution and Overflow attacks [EH]
– Check size of XML document before loading into parser processing
– When using signatures ensure the entire document is signed
– Always validate XML message against hardened schema
– Always use XSD and not DTD for schema definition
– Do not trust that sender has pre-valdiated XML, the recipient should always validate the XML message

SAMPLES

REFERENCES

[EH] “Configuring SAX processors for Secure Processing” by Elliotte Harold
http://www.ibm.com/developerworks/xml/library/x-tipcfsx.html

[BH] “A Taxonomy of Attacks against XML Signatures & Encryption” by Brad Hill
http://www.isecpartners.com/files/iSEC_HILL_AttackingXMLSecurity_Handout.pdf

A6- IDENTITY MISUSE

From a security perspective, identity is the basis of all control. Identity is literally the key that binds the subject (users, clients, service requesters) to the objects (service providers, servers, JNDI trees, etc). Yet applications rely on username/password combinations that have proven as easy targets by phishers and other attackers. Identity is transmitted via claims and assertions in web services. Identity is consumed by services to make three main types of decisions

- Routing decisions
- Business logic decisions
- Access control decisions

But many web services lack identity protection, one study showed that 72% of Enterprise Service Bus applications did not have strong identity protections [NC], and those that do often combine all of this into one “magic” token to rule them all.

ENVIRONMENTS AFFECTED

All web service frameworks are vulnerable to identity misuse.

VULNERABILITY

Identity information is a gateway vulnerability, breaking the account is often not the target the attacker seeks, but rather the data and functionality that the broken account affords. Identity information is misapplied and misused in the following ways:

- Tokens sent in the clear – identity tokens sent in the clear are vulnerable to replay, disclosure
- Token tampering – unsigned tokens are vulnerable to replay, substitution and elevation of privilege attacks
- Token bloat – Many organizations’ tokens disclose way too much information – tokens should only carry the attributes needed to accomplish a specific task.
- Weak protection schemes
- Weak authorization – the strongest token in the world is useless if the authorization logic that enforces policy is suspect
- Unscoped tokens – global tokens that are issued by identity providers and may be used across many applications (see Google SSO hole example below)
- Service providers use users identity information without their consent for profiling, demographic, marketing and other purposes. Sometimes, large histories are compiled with sensitive user data being tracked

A recent example was the hole in Google’s SSO service, here is Kim Cameron’s annotation of the CERT notice

“ A malicious service provider might have been able to access a user Google Account or other services offered by different identity providers. [US-CERT actually means 'by different service providers' - Kim] Google has addressed this issue by changing the behavior of their SSO implementation. Administrators and developers were required to update their identity provider to provide a valid recipient field in their assertions. http://www.identityblog.com/?p=1011

VERIFYING SECURITY

The goal is to verify that the application uses strong identity mechanisms, protects them for communication and routing, and enforces application authorization based on the strongly asserted claims.

Automated approaches: Vulnerability scanning tools will have difficulty identifying which identity technologies are being used. Likewise most static analysis tools do not know which identity technologies are used and how they are enforced

Manual approaches: Code review can check for strong identity usage and how the tokens are applied. Penetration testing can also verify that token manipulation and authorization circumvention is possible.

PROTECTION

A mix of technologies are required to beef up application identity security:

- Identity tokens, assertions, and claims must be hashed signed to deal with tampering,substitution and man in the middle attacks
- Sensitive information in the identity token should be encrypted at the message level, for example with XML Encryption
- The recipient must verify signature and the signer for authorization purposes
- Some applications that have replay and timing concerns should use a nonce fro replay and timestamp for message freshness and expiration
- Rely on open standards like SAML, WS-Security, and Information Cards

SAMPLES

REFERENCES

[NC] “Market Analysis – Enterprise Service Bus”, Network Computing http://www.networkcomputing.com/channels/appinfrastructure/showArticle.jhtml?articleID=181501609&pgno=8

“The Laws of Identity” by Kim Cameron
http://www.identityblog.com/?p=354

“A Primer on User Identification” by Stefan Brands
http://www.idtrail.org/files/Brands-a_primer-on_user_identification.pdf

"Debunking SAML Myths and Misunderstandings" by Frank Cohen
http://www.ibm.com/developerworks/xml/library/x-samlmyth.html

A7 –WEAK TOKENS

Proper token selection, usgae and token verification is critical to web services security. Flaws in this area most frequently involve the failure to protect credentials, tokens, claims, and assertions through their lifecycle. These flaws can lead to the hijacking of user or administrative accounts, undermine authorization and accountability controls, replay attacks, and cause privacy violations.

ENVIRONMENTS AFFECTED

All web services frameworks are vulnerable to authentication and token flaws.

VULNERABILITY

In SOAP and REST style web services there is no default authentication, messages are typically sent in XML over HTTP and contain nothing that can be used to perform authentication. Further, simply applying general purpose security standards like WS-Security is not adequate, the WS-Security Username token may pass the user's password in plaintext form. For example:

```
        <wsse:Username>Joe</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">MyPassword</wsse:Password>
```

The next step beyond Username Token with Password in cleartext is to look at hashing the password

```
        <wsse:Username>Joe</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest">E9rKWg/JSBzmaQufwyf0BRjcu3w=</wsse:Password>
```

This token is marginally stronger, but also lacks a timestamp and nonce so may be vulnerable to message replay and other attacks. Further, if the password is hashed, its likely there is a cleartext password sitting somewhere in the system that generated it.

Some web services apply security to the request but not to the response leaving the recipient's anonymous attack surface exposed and potentially disclosing sensitive data. Finally, signing the request and response message does not ensure resilience to replay attacks or other attacks that rely on message freshness and expiration windows.

In general, session management is less of an issue in web services that are typically (but not always) stateless. We leave this discussion aside, but if you are working with stateful web services, please review the OWASP Top Ten 2007 A7 - Broken Authentication and Session Management.

VERIFYING SECURITY
The goal is to verify that the service properly authenticates requests and responses and properly protects identities and their associated credentials, tokens, claims and assertions.

Automated approaches: Vulnerability scanning tools may be able to identify web services that are open to anonymous requests and responses, but beyond this have a very difficult time detecting vulnerabilities in custom authentication and session management schemes. Static analysis tools may be able to detect some authentication and token flaws.

Manual approaches: Code review and testing, especially in combination, are quite effective at verifying that authentication, token usage, and ancillary functions are all implemented properly. An architecture and policy review is necessary to find the right mix of security mechanism in a web service.

PROTECTION
Web service authentication relies on secure communication channel, and verifiable credentials, tokens, claims, and assertions. For example, one common Web service vulnerability is where the service requester signs the message, and the service provider checks to see if the message is signed before granting the request, however, the provider must also check that it trusts the signer!

Preventing authentication and weak token flaws takes careful planning. Among the most important considerations are:

* First ensure that SSL is the only option for all authenticated parts of the service communications (see A9 – Insecure Communications) and that all credentials, tokens, claims, and assertions are sent in signed, hashed and encrypted form (see A8 – Insecure Cryptographic Storage). Ensure that these are then verified.

* Limit or rid your code of custom authentication and token schemes, such as <username>joe</username> <password>hard2guess</password> type functionality, passing system user like "MQUSER" or home grown single-sign on functionality.

* Use a verifiable token such WS-Security with SAML, Kerberos or X.509 token to hash, sign, and encrypt the requests and responses.

* Use nonce, timestamp, and created/expired timestamp data to prevent session replay and other timing attacks. Ensure that this nonce and timestamp data is hashed and signed, see previous point

* Do not allow the login process to start from an unencrypted request or response. Always start the login process from an encrypted request or response with a fresh or new session token to prevent credential or token stealing attacks

* Consider regenerating a new token upon successful authentication or privilege level change.

* Do not rely upon spoofable credentials as the sole form of authentication, such as IP addresses or address range masks, DNS or reverse DNS lookups, referrer headers or similar

SAMPLES

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6145

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6229

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6528

REFERENCES

CWE: CWE-287 (Authentication Issues), CWE-522 (Insufficiently Protected Credentials), CWE-311
(Reflection attack in an authentication protocol), others.

WASC Threat Classification:
http://www.webappsec.org/projects/threat/classes/insufficient_authentication.shtml
http://www.webappsec.org/projects/threat/classes/credential_session_prediction.shtml
http://www.webappsec.org/projects/threat/classes/session_fixation.shtml

OWASP Guide, http://www.owasp.org/index.php/Guide_to_Authentication

OWASP Code Review Guide,
http://www.owasp.org/index.php/Reviewing_Code_for_Authentication

OWASP Testing Guide, http://www.owasp.org/index.php/Testing_for_authentication

RSNAKE01 - http://ha.ckers.org/blog/20070122/ip-trust-relationships-xss-and-you

A8 – INSECURE CRYPTOGRAPHIC STORAGE

Protecting sensitive data with cryptography has become a key part of most secured applications. Simply failing to encrypt sensitive data is very widespread. However simply using cryptography is not a panacea, applications that do encrypt frequently contain poorly designed cryptography, either using inappropriate ciphers or making serious mistakes using strong ciphers. These flaws can lead to disclosure of sensitive data and compliance violations. Cryptography warrants special attention in web services because the primary web service security mechanisms, such as WS-Security and SAML, directly rely on cryptography.

ENVIRONMENTS AFFECTED
All web services frameworks are vulnerable to insecure cryptographic storage.

VULNERABILITY
Preventing cryptographic flaws takes careful planning. The most common problems are:

* Not encrypting sensitive data

* Using home grown algorithms

* Insecure use of strong algorithms

* Continued use of proven weak algorithms (MD5, SHA-1, RC3, RC4, etc…)

* Hard coding keys, and storing keys in unprotected stores

* Misuse and mis-adaptation of web service security mechanisms (WS-Security, SAML, XML Signature, etc.)

VERIFYING SECURITY
The goal is to verify that the application properly encrypts sensitive information. This includes message exchanges and storage.

Automated approaches: Vulnerability scanning tools cannot verify cryptographic usage for message exchange and storage at all. Code scanning tools can detect use of known cryptographic APIs and Web service security standards, but frequently cannot detect if they are being used properly or if the encryption is performed in an external component.

Manual approaches: Like scanning, testing cannot verify cryptographic usage for message exchange and storage. Code review is the best way to verify that an application encrypts sensitive data and has properly implemented the mechanism and key management. This may involve the examination of the configuration of external systems in some cases.

PROTECTION
The most important aspect is to ensure that everything that should be encrypted is actually encrypted. Then you must ensure that the cryptography is implemented properly. As there are so many ways of using cryptography improperly, the following recommendations should be taken as part of your testing regime to help ensure secure cryptographic materials handling:

* Do not create cryptographic algorithms. Only use approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.

* Do not use weak algorithms, such as MD5 / SHA1. Favor safer alternatives, such as SHA-256 or better

* Generate keys offline and store private keys with extreme care. Never transmit private keys over insecure channels

* Ensure that infrastructure credentials such as database credentials or MQ queue access details are properly secured (via tight file system permissions and controls), or securely encrypted and not easily decrypted by local or remote users

* Ensure that encrypted data stored on disk is not easy to decrypt. For example, database encryption is worthless if the database connection pool provides unencrypted access.

* Do not roll your own Web service mechanisms to package the crypto, use standards for packaging crypto such as WS-Security and SAML

* Under PCI Data Security Standard requirement 3, you must protect cardholder data. PCI DSS compliance is mandatory by 2008 for merchants and anyone else dealing with credit cards. Good practice is to never store unnecessary data, such as the magnetic stripe information or the primary account number (PAN, otherwise known as the credit card number). If you store the PAN, the DSS compliance requirements are significant. For example, you are NEVER allowed to store the CVV number (the three digit number on the rear of the card) under any circumstances. For more information, please see the PCI DSS Guidelines and implement controls as necessary.

SAMPLES
* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6145
* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1664
* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-1101 (True of most Java EE servlet containers, too)

REFERENCES
* CWE: CWE-311 (Failure to encrypt data), CWE-326 (Weak Encryption), CWE-321 (Use of hard-coded
cryptographic key), CWE-325 (Missing Required Cryptographic Step), others.
* WASC Threat Classification: No explicit mapping
* OWASP, http://www.owasp.org/index.php/Cryptography
* OWASP Guide, http://www.owasp.org/index.php/Guide_to_Cryptography
* OWASP, http://www.owasp.org/index.php/Insecure_Storage
* OWASP, http://www.owasp.org/index.php/How_to_protect_sensitive_data_in_URL’s
* PCI Data Security Standard v1.1,
https://www.pcisecuritystandards.org/pdfs/pci_dss_v1-2.pdf
* Bruce Schneier, http://www.schneier.com/
* CryptoAPI Next Generation, http://msdn2.microsoft.com/en-us/library/aa376210.aspx
* "Hole in Google SSO Service", http://www.identityblog.com/?p=1011

A9 – INSECURE COMMUNICATIONS

Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications. Encryption (usually SSL) must be used for all authenticated connections, especially Internet-accessible web services, but backend connections as well. Otherwise, the application will expose sensitive data such as an authentication or session token. In addition, encryption should be used whenever sensitive data, such as credit card or health information is transmitted. Applications that fall back or can be forced out of an encrypting mode can be abused by attackers.

The PCI standard requires that all credit card information being transmitted over the Internet be encrypted.

ENVIRONMENTS AFFECTED
All web services frameworks are vulnerable to insecure communications.

VULNERABILITY
Failure to encrypt sensitive communications means that an attacker who can sniff traffic from the network will be able to access the conversation, including any credentials or sensitive information transmitted. Consider that different networks will be more or less susceptible to sniffing. However, it is important to realize that eventually a host will be compromised on almost every network, and attackers will quickly install a sniffer to capture the credentials of other systems.

For communications that cross security policy domains, such as B2B web services, using SSL is critical, as they are very likely to be using insecure networks to access applications. Because HTTP includes authentication credentials or a session token with every single request, all authenticated traffic needs to go over SSL, not just the actual login request.

Encrypting communications with backend servers is also important. Although these networks are likely to be more secure, the information and credentials they carry is more sensitive and more extensive. Therefore using SSL on the backend is quite important.

Encrypting sensitive data, such as credit cards and social security numbers, has become a privacy and financial regulation for many organizations. Neglecting to use SSL for connections handling such data creates a compliance risk.

Finally, web services security tokens like SAML and WS-Security tokens are sent along with web service request and responses, so if they are sent over an unencrypted communications channel like HTTP then they are visible to the attacker who can sniff traffic. The security token may grant access to the attacker for a variety of resources on the system not just the service it was stolen from.

VERIFYING SECURITY
The goal is to verify that the application properly encrypts all authenticated and sensitive communications.

Automated approaches: Vulnerability scanning tools can verify that SSL is used on the Web service provider, and can find many SSL related flaws. However, these tools cannot easily check if the response from the web services is using SSL, the request and reponse may very well use different protocols especillay in asynchronous message exchanges, the tools do not have access to backend connections and cannot verify that they are secure. Static analysis tools may be able to help with analyzing some web services, and their calls to backend systems, but probably will not understand the custom logic required for all types of systems.

Manual approaches: Testing can verify that SSL is used and find many SSL related flaws on the requests to Web Service providers, but the automated approaches are probably more efficient. Code review is quite efficient for verifying the proper use of SSL for all backend connections. Always remember to verify SSL use throughout the Web service lifecycle. Most web services use a Request-Response model, and often the most sensitive data is returned in the Response, so only checking Request to the Service Provider is insufficient. All Message Exchanges in the web service, i.e. the Request and Response must be verified.

PROTECTION
The most important protection is to use SSL on any authenticated connection or whenever sensitive data or tokens are being transmitted. There are a number of details involved with configuring SSL for web services properly, so understanding and analyzing your environment is important.

* Use SSL for all connections that are authenticated or transmitting sensitive or value data, such as tokens, credit card details, health and other private information

* Ensure that communications between infrastructure elements, such as between web services and database systems, are appropriately protected via the use of transport layer security or protocol level encryption for credentials and intrinsic value data

* Use SSL for the entire message exchange, i.e. request and response. Only protecting the logon credentials is insufficient because data and session information must be encrypted too.

* Under PCI Data Security Standard requirement 4, you must protect cardholder data in transit. PCI DSS compliance is mandatory by 2008 for merchants and anyone else dealing with credit cards. In general, client, partner, staff and administrative online access to systems must be encrypted using SSL or similar.
For more information, please see the PCI DSS Guidelines and implement controls as necessary

* Configure the transport to use SSL. For example in Apache Axis2 axis2.xml for a TransportSender, the service requester's request and the service provider response generators.

```
 <transportSender name="https"
class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
      <parameter name="PROTOCOL">HTTP/1.1</parameter>
      <parameter name="Transfer-Encoding">chunked</parameter>
   </transportSender>
```

and the Transport Receiver, the service provider's consumer and service requesters' consumers of inflow messages:

```
 <transportReceiver name="https"
class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
..
   </transportReceiver>
```

* Verify that the SSL is configured at deployment and run time by sniffing network traffic

* Consider authenticating the channel in addition to encrypting it, by using TLS.

SAMPLES
•• http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6430
•• http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4704
•• http://www.schneier.com/blog/archives/2005/10/scandinavian_at_1.html
REFERENCES
•• CWE: CWE-311 (Failure to encrypt data), CWE-326 (Weak Encryption), CWE-321 (Use of hard-coded
cryptographic key), CWE-325 (Missing Required Cryptographic Step), others.

WASC Threat Classification: No explicit mapping
•• OWASP Testing Guide, Testing for SSL / TLS, https://www.owasp.org/index.php/Testing_for_SSL-TLS
•• OWASP Guide, http://www.owasp.org/index.php/Guide_to_Cryptography
•• Foundstone - SSL Digger, http://www.foundstone.com/index.htm?subnav=services/navigation.htm&subcontent=/services/overvie
w_s3i_des.htm
•• NIST, SP 800-52 Guidelines for the selection and use of transport layer security (TLS) Implementations,
http://csrc.nist.gov/publications/nistpubs/800-52/SP800-52.pdf
•• NIST SP 800-95 Guide to secure web services,
http://csrc.nist.gov/publications/drafts.html#sp800-95
•• OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI
Apache Axis2 Transports - http://ws.apache.org/axis2/1_4/http-transport.html

A10 – Web Threats

Summary of OWASP top ten threats – Injection flaws, insecure direct object reference, …

ENVIRONMENTS AFFECTED

VULNERABILITY

VERIFYING SECURITY

PROTECTION

SAMPLES

REFERENCES