Testing i bones pràctiques

Josep Vañó Chic

PID_00208408

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Per a desenvolupar aplicacions segures s'ha de tenir en compte la seguretat. Això implica fer una sèrie de tasques específicament enfocades a la seguretat i que estan integrades en el cicle de vida del desenvolupament de programari segur. A més a més, és important de conèixer les tècniques de codi segur i les bones pràctiques perquè les aplicacions tinguin un nivell de seguretat òptim.

En la fase de desenvolupament s'ha d'incorporar la revisió de codi segur. La revisió de codi permet trobar i solucionar un gran nombre de problemes de seguretat abans d'entrar en la fase de proves. A més a més, aquesta revisió s'ha de fer cada vegada que hi ha un canvi significatiu.

En la fase de proves s'han de dur a terme específicament proves de seguretat, les quals es fan per a validar les mitigacions dissenyades en el modelatge d'amenaces. A diferència de les proves funcionals, les proves de seguretat miren de detectar si, per exemple, es pot suplantar la identitat d'un altre usuari o si es poden manipular les dades, que no es pugui accedir a dades a les quals no s'ha de tenir accés, que no es pugui tenir més privilegis a causa de l'ús maliciós de l'aplicació, que no es produeixin desbordaments. És a dir, en aquesta fase es tracta de pensar com un *hacker* per a intentar trobar vulnerabilitats per on es pugui atacar l'aplicació.

Les tècniques de codi segur i les bones pràctiques són un conjunt de consideracions que s'han de tenir en compte en la programació de codi segur, les quals és convenient d'aplicar i tenir en compte per a mitigar les amenaces tant com sigui possible.

Objectius

En acabar la lectura d'aquest mòdul, heu d'haver assolit les competències següents:

1. Conèixer les tècniques de codi segur bàsiques sobre criptografia, *SQL injection* i *cross site scripting*.

2. Saber les tasques que s'han de fer en la revisió de codi en una programació de codi segur.

3. Conèixer les tasques que s'han de dur a terme en les proves de seguretat.

4. Entendre les bones pràctiques en la programació de codi segur.

1. Tècniques de codi segur

Els tipus d'atacs que poden patir les aplicacions són molt diversos, i per a mitigar les amenaces s'han d'utilitzar tècniques de codi segur. Aquest apartat se centra a fer una aproximació en tres àmbits en concret: la criptografia, l'*SQL injection* i el *cross site scripting*.

1.1. Criptografia

En moltes aplicacions és necessari emmagatzemar i recuperar informació confidencial com, per exemple, contrasenyes, cadenes de connexió i altres dades confidencials a les quals no han de tenir accés altres usuaris o aplicacions. Aquesta característica es pot aconseguir amb solucions criptogràfiques estàndard.

Cada plataforma (MS Windows, Linux, Mac OS, etc.) proporciona alguna solució per a resoldre aquest problema, com, per exemple:

> CryptoAPI, CAPICOM, DPAPI, WebCryptoAPI, Keychain Services API on Mac OS, GNOME-Keyring, KWallet.

Es tracta de biblioteques que poden utilitzar els desenvolupadors en les aplicacions i que permeten als usuaris crear i intercanviar documents i altres dades en un entorn segur, sobretot en els mitjans de comunicació no segurs com Internet.

S'ha de tenir en compte que l'ús de la criptografia s'ha de considerar com una capa dins de la defensa en profunditat, és a dir, és convenient afegir més elements de seguretat als fitxers o dades amb informació sensible, com, per exemple, l'ocultació i els privilegis d'accés.

En la fase de disseny s'ha de definir en quins punts s'ha d'utilitzar la criptografia i quin tipus de criptografia és el més adequat en funció del tipus d'amenaça. A continuació mostrarem algunes de solucions que es poden utilitzar en funció del tipus d'amenaça:

Amenaça	Tècnica d'encriptació	Exemples d'algoritmes
Information disclosure	Criptografia simètrica	RC2, RC4, DES, 3DES, AES
Tampering	Funcions *hash*, signatura digital	SHA-1, SHA-256, SHA-384,SHA-512, MD4, MD5, HMAC, RSA *digital signatures*, DSS *digital signatures*, XML Dsig

Amenaça	Tècnica d'encriptació	Exemples d'algoritmes
Spoofing	Autenticació de dades	Certificats de clau pública i signatura digital

1.2. *SQL injection*

Una de les vulnerabilitats que ocupen els primers llocs del *top* 10 és l'*SQL injection*. Es tracta d'una vulnerabilitat fàcil d'explotar, i com que la finalitat que té és accedir a la base de dades, l'impacte pot ser molt greu. S'ha de tenir en compte que per a fer aquest tipus d'atac no es necessita tenir coneixements de la tecnologia, ni de la seguretat, ni eines especials, sinó simplement introduir una sèrie de caràcters en el navegador quan s'accedeix a un formulari d'un web.

Pàgina web

Podeu veure el *top* 10 de les vulnerabilitats a l'adreça següent:

The Open Web Application Security Project (OWASP).

Suposem que en algun punt d'un formulari d'un web es demana el codi de client per a accedir a les dades del client. Un cop introduït aquest codi, l'aplicació per a obtenir les dades en la base de dades crea una sentència SQL com la següent:

```
String sql = "SELECT * FROM clients WHERE CodiClient = ' " + pCodiClient + "'"
```

en què la variable *pCodiClient* conté el valor del codi que s'ha introduït en el formulari. Si el valor *pCodiClient* = A1234, la sentència SQL quedarà de la manera següent:

```
SELECT * FROM clients WHERE CodiClient = 'A1234'
```

Aquesta sentència retornarà concretament les dades del client A1234.

Però si l'usuari introdueix:

```
Codi Client: A1234' OR 1=1 - -
```

en aquest cas la sentència quedarà construïda de la manera següent:

```
SELECT * FROM clients WHERE CodiClient = 'A1234' OR 1=1 - -  '
```

Amb aquesta sentència, la base de dades retornarà les dades de tots els clients, ja que sempre es complirà la condició. Cal tenir en compte que, per exemple, en les bases dades de Microsoft SQL i Oracle Database, els dos guions (- -) s'interpreten de manera que tot el que hi ha a continuació de la mateixa línia es tracta d'un comentari, i per tant no executa.

Com es pot observar, per culpa de la vulnerabilitat en l'entrada d'informació es pot provocar l'alteració de la funcionalitat del formulari i obtenir informació de manera malintencionada. Suposem un altre exemple en què es demana un usuari i contrasenya per a accedir a l'aplicació. La consulta SQL per a verificar les dades podria ser de la manera següent:

```
"SELECT count(*) FROM usuaris WHERE user = '" + pUser + "' AND password = '" + pPassword + "'"
```

En una entrada normal, en què, per exemple, l'usuari sigui *admin* i la contrasenya sigui *mykey*, es construiria la consulta següent:

```
SELECT count(*) FROM usuaris WHERE user = 'admin' AND password = 'mykey'
```

En aquest cas, si en l'entrada d'usuari s'introdueix la sentència es construirà de la manera següent:

```
admin'  - -
```

```
SELECT count(*) FROM usuaris WHERE user ='admin'  - - ' AND password = ''
```

Com es pot observar, la consulta no verificarà la contrasenya, ja que la base de dades interpretarà la part `' AND password = ''` com un comentari, i en conseqüència s'hi podrà accedir com a usuari *admin* sense saber la contrasenya i així suplantar la identitat, amb tot el perill que això comporta.

Un cas similar és introduir:

```
' OR 1=1  - -
```

En aquest cas també es podria eludir l'autenticació ja que la sentència construïda seria de la manera següent:

```
SELECT count(*) FROM usuaris WHERE user = '' OR 1=1  - - ' AND password = ''
```

En l'exemple següent s'accediria a l'aplicació suplantant la identitat d'algun usuari:

```
' OR 1=1 AND user  > 'A'   - -
```

```
= '' SELECT count(*) FROM usuaris WHERE user = '' OR 1=1  AND user > 'A'  - - ' AND password
```

Amb l'*SQL injection*, no tan sols es pot obtenir informació o eludir l'autenticació sinó que també es pot inserir informació, ja que si s'introdueix:

```
' OR 1=1 ; INSERT  INTO usuaris (user, password)  VALUES ('josep' , 'mykey' )   - -
```

la sentència construïda serà de la manera següent:

```
SELECT count(*) FROM usuaris WHERE user = '' OR 1=1 ;
INSERT  INTO usuaris (user, password)  VALUES ('josep' , 'mykey' )   - - ' AND password = ''
```

El fet d'haver introduït un punt i coma fa que el que segueix a continuació s'interpreti com una nova sentència, la qual també s'executaria; en aquest exemple, en la segona sentència l'atacant inseriria un usuari en la base de dades. Això implica que l'atacant podria accedir a l'aplicació amb l'usuari que ha creat.

Com s'ha pogut observar en aquests exemples, amb la injecció de codi es poden alterar les sentències SQL, amb tot el que això pot comportar, com, per exemple, extreure, inserir i modificar dades o fins i tot executar ordres en el servidor de base de dades.

Per a evitar que l'aplicació pugui ser atacada amb *SQL injection* hi ha diverses opcions, tot i que no totes ofereixen un grau de seguretat òptim. A continuació descriurem les solucions possibles:

1) Filtrar l'entrada de dades

Hi ha dues maneres de filtratge: les llistes blanques (*white lists*), en les quals es bloqueja tot excepte el que es permet explícitament en la llista, i les llistes negres (*black lists*), les quals permeten deixar-ho passar tot excepte un conjunt de caràcters definits que poden causar problemes. Les llistes poden ser complicades de definir i mantenir, i tampoc no asseguren del tot que siguin efectives; per exemple, l'apòstrof s'utilitza en la gramàtica de diversos idiomes, i també es fa servir en alguns noms, per exemple *O'Connor*. Tot i així, aquest mètode d'utilitzar llistes blanques o negres és convenient de fer-lo servir, però s'ha de fer juntament amb altres mecanismes de defensa, que formen part de la defensa en profunditat, per a evitar l'*SQL injection*.

Un altre problema que presenta el sistema de filtratge en aquesta vulnerabilitat és que filtrant o convertint els apòstrofs no soluciona l'*SQL injection* en els paràmetres numèrics.

Suposem la consulta SQL següent:

```
SELECT nif, nom FROM clients WHERE codi = 43000001
```

L'usuari podria introduir el següent:

```
430000001 AND 1=0 UNION SELECT  user, password FROM usuaris
```

La construcció de la consulta resultaria de la manera següent:

```
SELECT nif, nom FROM clients WHERE codi = 43000001 AND 1=0 UNION SELECT  user, password FROM usuaris
```

El resultat d'aquesta consulta seria la llista d'usuaris i les seves contrasenyes.

2) Construcció de sentències SQL segures

La manera més segura d'evitar l'*SQL injection* és amb les sentències parametritzades (*parameterized commands*). Tot i que la manera d'indicar el paràmetre pot variar en funció de la base de dades, un exemple d'aquestes sentències pot ser el següent:

```
SELECT count(*) FROM usuaris WHERE user = ?  AND password = ?
```

En aquest cas, la construcció de la sentència no es fa per codi en l'aplicació sinó que la fa el motor de la base de dades. En funció del llenguatge de programació, les instruccions i la manera de passar els paràmetres poden variar, però en qualsevol cas no es fa en el codi de programació a força de concatenar cadenes i valors, sinó que es passen els valors dels paràmetres a la base de dades i és la base de dades la que construeix la sentència.

1.3. *Cross site scripting* (XSS)

Aquesta vulnerabilitat també és una de les més esteses. En aquest cas, l'atacant injecta codi que el navegador pot interpretar, sia amb marques HTML, amb JavaScript, etc.

Hi ha dos tipus de vulnerabilitats XSS:

- Persistent. En aquest cas, l'atacant aconsegueix gravar dades en el servidor, de manera que, quan un usuari accedeix al web, és víctima de l'XSS.

- No persistent. Aquest tipus es presenta quan un usuari accedeix a un web amb un URL especialment modificat en el qual hi ha els paràmetres de l'atac.

Un exemple de la manera com es pot fer aquest atac pot ser el següent:

> Suposem un web en què es pot escriure text; per exemple, un llibre de visites o un fòrum. Si en el text escrivim `<script> alert ('hello world'); </script>`, el navegador ho interpretarà com una ordre i ho executarà fent aparèixer un missatge, en aquest cas inofensiu. El problema està en el fet que en lloc de l'ordre *alert* es poden injectar altres ordres amb fins maliciosos.

Protegir-se de l'XSS és realment complicat, ja que no hi ha una solució universal. Un dels documents interessants per a prevenir l'XSS és la guia *XSS Filter Evasion Cheat Sheet* publicada per The Open Web Application Security Project (OWASP).

Com a mínim, per a prevenir d'aquest atac, en el disseny de l'aplicació es pot fer el següent:

1) Validar totes les entrades en els formularis: verificar que el tipus de dades i la longitud de cada camp es correspon amb el que s'espera i filtrar caràcters especials que puguin resultar perillosos.

2) Filtrar determinades paraules com SCRIPT, OBJECT, APPLET, EMBED, FORM.

Una bona pràctica en el filtratge és aplicar el filtratge basat en llistes blanques, de manera que es denega tot excepte allò que de manera explícita es vol permetre. El que no s'ha de fer és filtrar a partir dels elements de perillositat que coneixem, ja que és molt difícil que els coneguem tots amb totes les variants i, a més a més, poden aparèixer elements i vectors d'atac nous, amb la qual cosa el filtratge no seria eficient.

2. Revisió de codi segur

La revisió de codi es fa en la fase de desenvolupament i permet trobar i solucionar un gran nombre de problemes de seguretat abans d'entrar en la fase de proves.

La revisió del codi s'ha de fer cada vegada que hi ha un canvi significatiu, en lloc d'esperar fins al final del projecte i fer la revisió en un sol cop. D'aquesta manera es pot centrar l'atenció en el que s'ha canviat, en lloc de trobar tots els problemes de seguretat a la vegada.

Hi ha eines que es poden utilitzar per a fer aquesta tasca, però sempre és necessari complementar-la amb una revisió feta per un tècnic. Les eines no entenen el context, que és important per a una revisió de codi segur. Tot i així, les eines són útils per a avaluar grans quantitats de codi i detectar possibles vulnerabilitats, però després ha de ser una persona la que ha de verificar els resultats i determinar si es tracta d'un problema real, si és explotable i valorar el risc que implica per a l'organització. A més a més, les eines pot ser que no detectin tots els problemes de seguretat i que, per tant, sigui necessària igualment una revisió humana.

L'equip de revisió

Hi ha diversos criteris sobre la manera de confeccionar un equip de revisió. Tot i així, la majoria d'aquests criteris coincideixen en el fet que l'idoni és que sigui un equip reduït.

Una de les modalitats és el que s'anomena *peer review*, en què la revisió la fan dos desenvolupadors que revisen el codi que han fet tots dos separadament.

Una modalitat similar és la formada pel desenvolupador i un o dos revisors destinats a aquesta tasca. El desenvolupador fa una primera revisió, i després la fan els revisors. En aquest punt, el desenvolupador col·labora explicant què fa el codi, sobretot en les seccions on el codi és més complex.

Segons la dimensió i el tipus del programari, l'equip està format pels rols següents:

- **Analista de negoci**. Descriu els objectius de seguretat des del punt de vista de negoci.

- **Arquitecte d'aplicacions**. Explica els objectius de seguretat des del punt de vista de l'arquitectura del programari i del sistema.

- **Desenvolupador**. Descriu els detalls del codi, revisa el codi i localitza els *bugs* o forats de seguretat.

- **Revisador**. Revisa el codi i localitza els *bugs*.

La modalitat que s'ha d'adoptar depèn del tipus i de la magnitud del programari, del tipus d'organització o de la dimensió de l'equip del departament, però en el que sí que coincideixen les diverses modalitats és que la de revisió de codi l'han de formar equips reduïts amb un màxim d'unes quatre persones, i si la dimensió del projecte ho requereix, s'han de crear diversos equips reduïts de revisió. A més a més, la formació d'aquests equips en el procés de revisió de codi permet a l'equip de desenvolupament compartir experiències i bones pràctiques de seguretat que poden prevenir problemes de seguretat en el futur.

2.1. Fases en la revisió de codi segur

- Identificar els objectius de la revisió.
- Fer una anàlisi preliminar.
- Fer una revisió de codi segur.

2.1.1. Identificar els objectius de la revisió

Per a saber què s'ha de revisar, s'han d'establir els objectius i les limitacions per a la revisió de codi, ja que per a poder fer una revisió s'ha de saber què s'està buscant. Si es comença la revisió sense establir objectius, augmenten les possibilitats de ser aclaparats pel codi i disminueixen les possibilitats de trobar problemes de seguretat.

S'ha de tenir en compte que en el modelatge d'amenaces s'han identificat i documentat les amenaces i a la vegada s'ha analitzat i elaborat un diagrama de flux de dades; per tant, si s'ha fet el modelatge d'amenaces, ja tenim el punt de partida. Cal recordar que el modelatge d'amenaces és un procés iteratiu i que durant la revisió de codi potser es trobaran noves amenaces. Així doncs, és molt important que les noves amenaces trobades en aquest procés de revisió s'incorporin en el modelatge d'amenaces per a tenir el model actualitzat i poder fer els controls posteriors, com, per exemple, en la fase de proves, d'una manera correcta.

A l'hora de determinar els objectius de la revisió hem de considerar el següent:

- Quines de les amenaces identificades en el modelatge d'amenaces afecten el codi que s'està revisant? En aquest punt es poden separar les amenaces que han estat mitigades i les que no ho han estat.

- Quins errors comuns de codificació poden ser presents en el codi que s'està revisant? És molt útil per a la revisió disposar d'una llista d'errors comuns per a ajudar a concentrar-se en punts en concret i a la vegada poder trobar

errors més fàcilment. Aquesta llista, òbviament, s'ha d'anar actualitzant amb els nous errors que es trobin i que siguin reiteratius o habituals.

2.1.2. Fer una anàlisi preliminar

En aquest punt es fa una anàlisi del codi per a trobar possibles problemes o punts febles de seguretat. Aquesta anàlisi preliminar es pot fer amb dos tipus d'anàlisi tenint en compte que una no exclou l'altra, de manera que es poden combinar tots dues:

- Revisió automàtica.
- Revisió manual.

1) Revisió automàtica

Una revisió automàtica feta amb una eina d'anàlisi estàtica pot donar com a resultant un conjunt de falses alarmes de seguretat i a la vegada pot ser que no detecti tots els problemes reals. Tot i així, també permet trobar problemes de seguretat que potser no es trobarien en una revisió manual.

2) Revisió manual

Tenint en compte que més endavant es farà una revisió de codi en matèria de seguretat d'una manera més exhaustiva, en l'anàlisi preliminar es tracta de fer una revisió que respongui, per exemple, a preguntes del tipus següent:

- Validació d'entrada. Es fa validació d'entrada? On es fa? En el client, en el servidor, o en tots dos llocs?

- Autenticació i autorització dels usuaris. Quin mecanisme s'utilitza en l'autenticació i autorització?

- Tractament d'excepcions. Hi ha àrees de codi amb un tractament d'excepcions massa dens o escàs?

- Codi complex. Hi ha àrees de codi complex?

2.1.3. Fer una revisió de codi segur

La frontera entre una revisió manual en una anàlisi preliminar i la revisió de codi segur no és una línia marcada d'una manera categòrica; de fet, es podria considerar com una revisió en espiral en què es comença per una visió general i es va profunditzant en la revisió.

S'ha de tenir en compte que en la revisió hi participa personal addicional a més de l'autor del codi, i fins i tot es pot donar la circumstància que l'autor del codi no hi sigui.

El primer que ha de fer l'equip de revisió és conèixer el flux bàsic de l'aplicació i les funcionalitats que té. Un cop se sap com funciona l'aplicació, cal fer referència al modelatge d'amenaces que s'ha creat prèviament i al diagrama de flux de dades. Això fa que la revisió es pugui enfocar a les parts de l'aplicació on la seguretat és més crítica.

Un cop es tenen classificades les parts de codi en funció del risc, cal prioritzar i aplicar un esforç proporcional al nivell de risc; per exemple, en una àrea de codi d'alt risc s'ha de fer una revisió detallada línia per línia, en un mòdul de menys risc es pot fer una revisió menys detallada i en una zona de poc risc es poden examinar només les crides a funcions que puguin comportar algun tipus de risc.

Per a fer la revisió s'han de combinar les tècniques següents:

- **Anàlisi del control del flux**. En aquesta anàlisi es revisen pas a pas les condicions lògiques, els condicionals i els bucles iteratius en el codi, tenint en compte les diverses branques que hi poden haver.

- **Anàlisi del flux de dades**. En aquesta anàlisi es rastregen les dades des dels punts d'entrada fins als punts de sortida; partint dels punts d'inici cal fer el recorregut fins al final del procés. Si un punt de partida té massa ramificacions, és convenient crear una nova ramificació com a punt de partida, però tenint en compte on té el punt de partida inicial. En l'anàlisi del flux de dades és necessari prestar especial atenció a les zones on s'analitzen dades i als punts on es poden enviar a diversos punts de sortida, i també si són tractades amb el nivell de confiança adequat.

Cal evitar el codi excessivament complicat. Com més complicat és el codi, més possibilitats hi ha que tingui errors de seguretat, i a la vegada, corregir codi excessivament complicat implica una alta probabilitat que s'introdueixin nous errors en els canvis que s'han fet.

En general, hi ha certs errors que cometen la majoria de programadors, i fins i tot és possible trobar patrons d'errors per a cada programador en concret.

Un aspecte que també s'ha de considerar en la revisió del codi són les crides a funcions no segures de les API; per exemple, funcions del llenguatge C com strcpy i strcat són funcions no segures ja que són susceptibles de provocar, si no es filtren correctament les dades, un *buffer overrun*. Així doncs, en la revisió

del codi s'han de canviar aquests tipus de funcions per les funcions equivalents i segures. En general s'ha d'evitar que es pugui produir cap mena de desbordament, ja que els *exploits* poden aprofitar qualsevol tipus de desbordament.

3. Proves de seguretat

Les proves de seguretat es fan per a validar les mitigacions dissenyades en el modelatge d'amenaces. A diferència de les proves funcionals, les proves de seguretat miren de detectar si, per exemple, es pot suplantar la identitat d'un altre usuari, si es poden manipular les dades, que no es pugui accedir a dades a les quals no s'ha de tenir accés, que no es pugui tenir més privilegis a causa de l'ús maliciós de l'aplicació, que no es produeixin desbordaments. És a dir, en aquesta fase es tracta de pensar com un *hacker* per a intentar trobar vulnerabilitats per on es pugui atacar l'aplicació.

Les proves de seguretat s'han de fer després de les proves funcionals. Un error funcional també pot representar un potencial problema de seguretat; per tant, primer s'han de resoldre els possibles problemes funcionals i després els de seguretat.

Aquesta fase s'ha de preveure des de l'inici del projecte i s'hi ha d'involucrar l'equip de proves des de la fase del disseny i el modelatge d'amenaces, i també s'han de revisar les especificacions de problemes de seguretat.

Per a fer les proves de seguretat s'ha de dur a terme el procés següent:

- Descompondre l'aplicació en components.
- Identificar les interfícies dels components.
- Classificar les interfícies per vulnerabilitats potencials.
- Determinar les estructures de dades utilitzades per cada interfície.
- Trobar problemes de seguretat mitjançant la injecció de dades transformades.

1) Descompondre l'aplicació en components

La llista de components en el sistema, els tipus d'amenaces per a cada component (STRIDE) i el risc d'amenaces (DREAD) són elements que prèviament s'han elaborat en el modelatge d'amenaces.

2) Identificar les interfícies dels components

Es tracta de determinar les interfícies exposades per a cada component, que poden estar exposades o no en el model d'amenaces. Tot i així, les interfícies exposades per a cada component s'han de trobar en les especificacions funcionals; en cas contrari, s'han d'obtenir amb la lectura del codi o s'ha de preguntar a l'equip de desenvolupament. En cas que les interfícies no s'hagin documentat abans, s'ha d'aprofitar aquest moment per a fer-ho.

A continuació mostrarem alguns exemples d'interfícies i tecnologies de transport:

- TCP i UDP *sockets*
- *Wireless data*
- NetBIOS
- *Mailslots*
- *Dynamic data exchange* (DDE)
- *Named pipes*
- *Shared memory*
- *The clipboard*
- *Local procedure call* (LPC) i *remote procedure call* (RPC) interfaces
- COM mètodes, propietats i esdeveniments
- *ActiveX controls* i applets
- EXE i DLL funcions
- HTTP requests and responses
- *Simple object access protocol* (SOAP) requests
- *Console input*
- *Command line arguments*
- *Dialog boxes*
- Fitxers
- *Database access technologies*, incloent-hi OLE DB i ODBC
- *Database stored procedures*
- *Environment variables*
- LDAP, active directory
- *Hardware devices*

3) Classificar les interfícies per vulnerabilitats potencials

La classificació inicial dels riscos ha de provenir del model d'amenaces; el pas següent, en cas que sigui necessari, és afegir més granularitat i precisió enfocades a fer les proves. La classificació ha d'estar ordenada segons el nivell de risc, d'aquesta manera es pot tenir una aproximació de la quantitat d'interfícies vulnerables que s'han de provar més a fons.

4) Determinar les estructures de dades utilitzades per cada interfície

El pas següent és determinar les dades que accedeixen a cada interfície. Es tracta de les dades que s'han de modificar per a exposar els errors de seguretat. A continuació mostrarem un exemple de fonts de dades segons la interfície:

Interfície	Dades
Sockets, RPC, *Named pipes*, NetBIOS	Dades que arriben per la xarxa.
Fitxers	El contingut dels fitxers.

Interfície	Dades
Active directory	Nodes en el directori.
HTTP *data*	HTTP capçaleres, entrades de formulari, *query strings*, etc.

Per a fer les proves de seguretat, s'ha de dur a terme la construcció dels casos de proves. L'enfocament STRIDE sobre els tipus d'amenaces en el modelatge d'amenaces ajuda a determinar els tipus de proves en cada cas.

A continuació mostrarem alguns exemples de tècniques de proves per a verificar les tècniques de mitigació:

Tipus d'amenaça	Tècniques de proves
Spoofing	• Intentar forçar l'ús de l'aplicació sense autenticació. • Es poden veure les credencials d'usuari en el transit de la xarxa o en un emmagatzematge persistent? • Els "*tokens* de seguretat", per exemple, es poden reproduir amb una *cookie* per a ometre la fase d'autenticació?
Tampering	• Intentar eludir l'autorització. • És possible manipular les dades i tornar a crear la funció resum (*rehash)* de les dades? • Crear funcions resum (*hash*) invàlides i signatures digitals i després verificar l'autenticitat.
Repudiation	• Hi ha condicions que eviten la creació d'un registre (*log*) o auditoria? • Es pot fer d'alguna manera que les dades del registre es gravin de manera incorrecta? Per exemple, afegint caràcters de salt de línia, retorn o de final de fitxer en una petició vàlida. • Hi ha algun tipus d'acció que ometi algun control de seguretat?
Information disclosure	• Intentar accedir a dades a les quals només s'hauria de poder accedir amb privilegis més elevats, tant de dades persistents en fitxers i en bases de dades com mitjançant el trànsit de la xarxa. Els *sniffers* de xarxa són una bona eina per a aquesta tasca. • Inspeccionar si s'han gravat dades sensibles en el disc; per exemple, en fitxers temporals, en *cookies*.
Denial of service	• Inundar de peticions el servidor per a intentar que quedi col·lapsat i no pugui donar servei. • Enviar dades amb formats o dimensions incorrectes, provoca que s'aturi el servei? • Quin punt afecten factors com, per exemple, poc espai disponible en el disc, capacitat de memòria, sobrecàrrega del processador o limitació de recursos en general, i provoquen que falli l'aplicació?
Elevation of privilege	• S'executen aplicacions o serveis del sistema amb privilegis més elevats dels necessaris? • Un procés, pot forçar, per exemple, la càrrega del *Command Shell* i executar processos amb privilegis més elevats?

Cada amenaça en el modelatge d'amenaces ha de tenir un pla de proves i ha de preveure que es dugui a terme una o diverses proves.

En cada test s'han de preveure els paràmetres que determinin l'èxit o no del test, i també la verificació de si la funció verificada ha fallat o no ha fallat.

5) Trobar problemes de seguretat mitjançant la injecció de dades transformades

En aquest cas es tracta d'introduir en cada via d'entrada caràcters, formats de dades i dades no previstes, dades transformades, dades fora de rang, etc., per a intentar que l'aplicació alteri el comportament que té o doni un error.

Les dades transformades són sovint utilitzades pels atacants per a provocar qualsevol tipus de desbordament i aprofitar-lo per a introduir un *exploit* mitjançant aquesta vulnerabilitat. A més a més dels que es poden fer provocant un desbordament, hi ha altres atacs que es poden rebre per les vies d'entrada de dades i formats no previstos, com, per exemple, l'*SQL injection* i el *cross site scripting*.

Per a fer aquest test cal preparar una bateria de dades, les quals s'han d'introduir per les vies d'entrada per a comprovar si l'aplicació fa correctament la validació de dades introduïdes. Aquesta bateria de dades ha de ser tan àmplia com sigui possible per a verificar que un atacant no pugui aprofitar aquests punts d'entrada.

4. Bones pràctiques

Hi ha tècniques eficaces que els desenvolupadors poden utilitzar per a prevenir els atacs. Algunes d'aquestes tècniques s'haurien d'utilitzar en desenvolupar qualsevol aplicació en qualsevol plataforma. Alguns atacs poden ser mitigats amb mètodes específics, alguns dels quals són els següents:

- Executar amb privilegis mínims.
- Reduir la superfície d'atac.
- Instal·lar les mínimes característiques per defecte.
- Validar l'entrada de l'usuari.
- Fer sevir la defensa en profunditat.
- Considerar els sistemes externs com a insegurs.
- No confiar en la seguretat ocultant dades.
- No revelar informació dels errors.
- No fer decidir els usuaris.
- Aprendre dels errors.
- Solucionar els problemes de seguretat de manera correcta.

1) Executar amb privilegis mínims

Executar estrictament, just amb els privilegis necessaris per a dur a terme la tasca o funcionalitat en qüestió.

Per exemple, si es produeix un error per desbordament, com en el cas d'un *buffer overrun*, i s'està executant en un context d'administrador, un *hacker* pot sobreescriure l'adreça de retorn i executar el codi maliciós amb privilegis d'administrador.

Certament, el més fàcil per als desenvolupadors és crear aplicacions amb permisos d'administrador perquè així no hi ha errors de permisos, però és una mala pràctica en la qual no s'ha de caure. També és una mala pràctica decidir donar permisos d'administrador temporalment amb la intenció de canviar després els privilegis pels que realment correspongui. Fer això comporta un alt risc, ja que per falta de temps, per oblit o per qualsevol altre circumstància pot ser que no es faci el canvi i es deixi la funció amb privilegis d'administrador.

2) Reduir la superfície d'atac

Els atacants sovint exploten les debilitats dels diversos serveis de la plataforma, com, per exemple, l'*index service* de l'*Internet information server* (IIS), i també les interfícies que exposen les aplicacions.

En aquest cas, una bona pràctica és la de desactivar tots els serveis que no s'utilitzin i limitar el nombre d'interfícies exposades per l'aplicació.

3) Instal·lar les mínimes característiques per defecte

Minimitzar la superfície d'atac també significa definir una instal·lació segura de l'aplicació. Com més característiques estan activades per defecte, més s'augmenta la possibilitat d'una violació de seguretat. Així doncs, és aconsellable activar el mínim de característiques possibles i assegurar-se que aquestes característiques són segures.

Es tracta, doncs, d'activar les característiques mínimes i adequades per a la majoria d'usuaris, i les funcions que s'utilitzin amb menys freqüència o les que utilitzin un grup reduït d'usuaris deixar-les desactivades per defecte per a reduir l'exposició de funcionalitats. Si una funció no s'està executant, no pot ser vulnerable a l'atac.

4) Validar l'entrada de l'usuari

Mai no s'ha de confiar en les dades introduïdes per l'usuari, i s'ha de considerar que tota entrada és sospitosa fins que no es demostri el contrari; per tant, l'aplicació ha de validar les dades introduïdes per l'usuari abans d'utilitzar-les.

També cal considerar la possibilitat que un *hacker* eludeixi la validació al costat del client d'una aplicació; per tant, la validació de dades no solament s'ha de fer en el punt d'entrada de dades sinó que també s'ha de fer en altres punts, com, per exemple, al costat del servidor o abans de gravar-les en la base de dades.

Hi ha diversos mètodes per a validar les dades i que a la vegada són compatibles entre si, és a dir, que se'n pot utilitzar més d'un per a validar una mateixa entrada:

- Acceptar valors vàlids i denegar la resta en lloc de denegar valors invàlids i acceptar la resta.
- Invalidar entrades que contenen caràcters o paraules clau que poden ser perillosos, com, per exemple, (`\`, `<`, `>`, `script`, `object`, `insert`).
- Invalidar entrades que excedeixin certa longitud.
- Restringir les entrades utilitzant controls de validació i expressions regulars.

5) Fer servir la defensa en profunditat

No s'ha de confiar en una única capa de defensa: amb només una capa de defensa no n'hi ha prou. Amb la construcció de múltiples capes de defensa es pot augmentar la seguretat d'un sistema.

Moltes aplicacions estan dissenyades i escrites de manera que confien totalment amb el tallafoc, però amb això no n'hi ha prou. El maquinari pot fallar, es pot desconfigurar, es pot haver configurat incorrectament, etc.; per tant, si només s'ha confiat en una sola capa de defensa, el risc de ser atacats amb èxit és molt elevat.

Encara que s'apliquin diverses capes de seguretat en l'àmbit de maquinari, tampoc no s'ha de deixar la responsabilitat de la seguretat als altres, és a dir, l'última capa de seguretat ha de ser en l'aplicació mateixa, fins i tot en cada capa de l'aplicació.

6) Considerar els sistemes externs com a insegurs

De les dades que es reben des d'un altre sistema no es té el control complet i podrien ser insegures i una font d'atac. Fins que no es pugui demostrar el contrari, totes les entrades externes són un atac potencial. Les dades d'entrada no solament poden provenir d'un usuari, sinó que també poden provenir d'un servidor extern, el qual pot ser un possible atacant.

En una arquitectura client-servidor, els clients poden ser redirigits de diverses maneres i de manera malintencionada cap a un altre servidor. Així doncs, en escriure codi al costat del client (*client-side code*) no s'ha de suposar que el client s'està comunicant amb el servidor correcte.

Des del punt de vista del servidor, tampoc no s'ha de suposar que les dades o peticions es reben des de la interfície d'un client web, perquè els atacants poden enviar dades malicioses eludint la interfície del client.

7) No confiar en la seguretat ocultant dades

A vegades hi ha la necessitat de guardar claus d'encriptació o un altre tipus de dades d'alt risc, i això es fa guardant aquesta informació de manera oculta. El problema està en el fet que en el moment en què es guarda informació en el disc, encara que sigui oculta, un *hacker* la pot trobar. De fet, els *hackers* saben que s'utilitza la tècnica d'ocultació de dades, i per tant busquen aquestes dades per la raó que les dades ocultes segur que són importants.

Ocultar dades és una bona defensa, però no ha de ser l'única que s'ha d'aplicar a aquestes dades. L'ocultació de dades només s'ha d'utilitzar com una petita part d'una defensa integral en l'estratègia de la defensa en profunditat.

8) No revelar informació dels errors

Si l'aplicació dóna un error, no s'ha de donar informació de l'error als usuaris finals. S'ha d'evitar donar informació, per exemple, de la instrucció, la línia del codi o el tipus d'error, és a dir, qualsevol informació tècnica. Per *informació* també s'hi entén donar explicacions de per què o com s'ha produït l'error.

Aquesta informació la pot aprofitar un atacant per tal de provocar l'error i atacar el sistema. Per a capturar informació dels errors es pot guardar aquesta informació en un fitxer de registre d'esdeveniments (*log*). El maneig d'errors ha de ser estructurat de tal manera que els detalls de l'error no es propaguin de tornada a l'usuari.

9) No fer decidir els usuaris

Hi ha moltes aplicacions en què l'usuari ha de prendre decisions de seguretat. S'ha de tenir en compte que la majoria d'usuaris no entenen en seguretat ni en qüestions tècniques del maquinari ni del programari, i a més a més no hi volen entendre. Els usuaris el que volen és que les seves dades i els ordinadors estiguin protegits sense haver de prendre decisions complexes. També cal recordar que, si a l'usuari se li donen diverses opcions per escollir, la majoria escolliran el botó per defecte, encara que no sàpiguen ben bé el que se'ls pregunta.

Tot i així, si s'opta perquè prengui alguna decisió, s'ha de tenir en compte que l'usuari no és cap tècnic en informàtica, i per tant la redacció de la pregunta ha de ser simple, fàcil d'entendre i sense tecnicismes; a més a més, no s'ha de saturar el quadre de diàleg amb explicacions innecessàries i, si és possible, s'ha d'afegir un botó d'ajuda on es puguin donar explicacions més detallades en un vocabulari entenedor per a l'usuari.

10) Aprendre dels errors

Si no s'aprèn dels errors, és molt probable que es cometi el mateix error en un futur.

A vegades un mateix problema de seguretat succeeix diversos cops. Si ja s'ha tingut l'experiència d'un problema en concret, com és que es torna a repetir el mateix problema?

Aprendre dels errors és un exercici, i cal tenir l'hàbit de documentar i actualitzar el modelatge d'amenaces, de crear una base de dades amb informació sobre les amenaces, de conscienciar a tot l'equip i de formular-se preguntes com les següents:

- Com es va produir l'error de seguretat?
- Es repeteix l'error en altres àrees del codi?
- Com es podria haver evitat l'error?
- Què es pot fer per a assegurar que aquest tipus d'error no torni a passar?
- És necessari actualitzar algun tipus d'eina o d'anàlisi?

11) Solucionar els problemes de seguretat de manera correcta

Si es troba una vulnerabilitat de seguretat en el codi o en el disseny, s'ha d'arreglar el problema i cercar problemes similars en l'aplicació. És molt probable que se'n trobin més de semblants i que el mateix problema també sigui en altres punts de l'aplicació.

De manera similar, si es troba un conjunt de defectes que segueixen un mateix patró, cal fer el passos necessaris per a crear mecanismes de defensa que redueixin o solucionin aquest tipus de problemes, en lloc de limitar-se a resoldre els problemes per parts.

Si es troba un problema de seguretat, d'entrada s'ha d'arreglar en el lloc més proper possible de la ubicació de la vulnerabilitat. Per exemple, si l'error és en una funció anomenada *DataControl* i la solució es pot fer en diversos punts del programa –per exemple, filtrant les dades a l'entrada de l'usuari, filtrant les dades en la lògica del programa o filtrant les dades dins de la funció–, on primer s'ha de solucionar és en la funció, ja que, si un atacant eludeix una part del programa i accedeix directament a la funció, el sistema continua essent vulnerable a ser atacat. Després també s'han d'arreglar les altres parts del programa perquè no arribin dades incorrectes a la funció.

Bibliografia

Howard, M.; LeBlanc, D. (2002). *Writing Secure Code* (2a. ed.). Redmond (Washington): Microsoft Press.

Microsoft (2013). *Testing for Securability* [en línia]. http://msdn.microsoft.com

The Open Web Application Security Project, OWASP (2013). http://www.owasp.org/

The Open Web Application Security Project, OWASP (2013). *Security Code Review in the SDL* [en línia]. https://www.owasp.org/index.php/Security_Code_Review_in_the_SDLC