UPPSALA UNIVERSITET CAMPUS GOTLAND
Institutionen för informatik och media
Utbildningsprogram: Kandidatprogram i Systemvetenskap (inriktning Programvaruteknik)
Kursansvarig lärare: Millan Lundgren

Datum: [2020-01-20]

Rapport om Säkerhetstestning
IT-säkerhet, 7,5 hp

Utvärdering av webbapplikation

Jonas Örnfelt

1 Problemformulering

Som konsult på företaget *CodeSec* har jag fått i uppdrag att utvärdera säkerheten hos eran webbapplikation. Utvärderingen ämnar studera dess uppbyggnad samt hur väl den behandlar data utifrån ett säkerhetsperspektiv.

1.1 Antaganden

Det första antagandet jag formulerar inför denna uppgift är definitionen av de tre kategorier som applikationens säkerhetsaspekter kommer delas in i. Kategorierna innefattar konfidentialitet, där applikationen bör sträva efter att inte tillgängliggöra information för obehöriga personer eller system. Tillgänglighet, där informationen bara görs åtkomlig för behöriga användare när det behövs. Riktighet, där information som visas är korrekt och uppdaterad.

Klassificering av säkerhetsaspekter baseras på mitt eget perspektiv på vad som är en allvarlig, men också rimlig konsekvens. Vidare antar jag här att jag har befogenhet att placera dessa utan att lägga större vikt i vad dess medelvärdiga konsekvensnivå kan tänkas vara.

Ett antagande kring verksamhetens sätt att jobba med ärenden görs även för att lättare kunna placera konsekvensgraden av förlust av digital information. Jag antar således att verksamheten inte arbetar med fysiska kopior av informationen som hämtas från databasen.

2 Resultat

2.1 Klassificering av data

Konsekvensnivå	Konfidentialitet	Riktighet	Tillgänglighet
Allvarlig 3	*Logins*		
Betydande 2	*Employees*	*Crime*	
Måttlig 1		*CrimeStatus, Pictures, Samples, Sequence*	
Ingen eller försumbar* 0		*Departments*	

Figur 1

Datatabellen *Logins* innehållandes inloggningsuppgifter för anställda klassificerar jag som en allvarlig säkerhetsaspekt inom kategorin konfidentialitet (se figur 1). Det kan skilja sig i konsekvensgrad beroende på vilken roll som kapade inloggningsuppgifter tillhör. Jag väljer att placera tabellen baserat på en högre roll, såsom Manager. En attack kan då potentiellt innebära förödande konsekvenser.

Datatabellen *Employees* klassificeras som betydande för konfidentialitet då det handlar om individers integritet samt kan leda till fientliga konsekvenser på organisations- och individnivå. Tabellen *Crime* som rimligtvis innehåller ärenden hade kunnat placeras i alla tre inriktningar. Här gäller det därför att se över konsekvensgraden och fatta beslut därefter. Ett ärendes konfidentialitet mynnar ut i informationen som kan erhållas vid en potentiell läckning.

Det är främst personuppgifterna av anmälaren som exempelvis kan hamna i fel händer i form av boven av miljöbrottet. När det kommer till tillgängligheten av den data tabellen innehåller kan detta påverka organisationen på ett ytterst ogynnsamt sätt. Om ärenden inte längre är åtkomliga behöver de som är ansvariga för ärendet förlita sig på fysiska uppgifter, vilket troligtvis är begränsat med tanke på att de främst hämtas via databasen.

Kategorin riktighet är den som har mest betydande konsekvenser och jag väljer att placera den på konsekvensnivån betydande. Placeringen motiveras med att ett ärendes uppgifter som inte stämmer kan ställa till med bekymmer på olika nivåer. Det är givet att fel plats, typ av brott och uppgifter om anmälaren kan leda till problem direkt i utredningen. Organisationen påverkas genom att de inte kan utföra sitt jobb och enstaka individer blir berörda ifall fel person kontaktas eller om fel plats undersöks.

Tabellen *Crimestatus* är en tabell som har en mindre påverkan på så sätt att den endast innehåller två värden som används för att sätta ärendestatus. Jag placerar denna på måttlig nivå inom riktighet. Om data i tabellen inte stämmer påverkas ärendens status-nivåer och kan leda till problem små som stora. Jag finner dock att tabellen skulle behöva innehålla fel data under en längre tid för att konsekvenserna skulle bli särskilt allvarliga.

Departments anser jag är den minst betydande tabellen och placeras därför på Ingen eller försumbar-nivån under riktighet. Tabellen innehåller, likt *Crimestatus* endast värden för att sätta korrekt avdelning på ärenden. Det denna tabell kan ställa till med är endast förvirring i organisationen, men detta är samtidigt något som är tillräckligt tydligt för att märkas omgående.

Pictures, *Samples och Sequence* är också tabeller som inte skulle ha en större påverkan. De två första är till för sparande av filer som är relevanta för ett visst ärende. *Sequence* används för att se till att varje nytt ärende får ett unikt ärendenummer. Dessa placeras under nivån måttlig inom inriktningen riktighet.

Placeringen stöds genom hur förvirring även här kan uppstå, men också ha negativa konsekvenser för organisation och individ. Om exempelvis

filer hamnar i fel ärenden skapas förvirring, vilket också kan leda till en mindre effektiv utredning eller ytterligare missuppfattningar.

Den sista tabellen som återfinns internt i databasen och ansvarar för nedsparandet av ärenden sätts på nivån betydande under riktighet med samma motivering som tabellen innehållandes ärendena i sig. Detta eftersom samma typ av uppgifter går förlorade vid en säkerhetsincident.

Vid första anblick kan det data som finns i ärende-tabellerna tänkas kunna delas upp i två olika tabeller där en innehåller de uppgifter som är känsliga och en innehållandes andra uppgifter. Dock skulle jag säga att detta blir en överflödig lösning då det skulle innebära ytterligare ett id-värde för att kunna koppla de två samman.

Vidare är också den information som skulle ingå i den andra tabellen bara ett par värden och dessa är bara id-värden som kopplar ärendet samman med övriga tabeller för att få fram ärendets uppgifter som mer berör verksamheten (som status och avdelning). Jag anser att hela tabellen bör krypteras istället för delas upp i detta fall.
Eftersom tabellen *Logins* innehåller så pass känsliga uppgifter är det angeläget att separera tabellen till en separat databas samt att kryptera innehållet. Personuppgifter kan även tänkas delas in i denna databas med krypterat innehåll. Detta för att skydda anställdas integritet och förhindra spridning av deras personliga uppgifter. Förslagsvis vill jag att inloggningsuppgifterna delas in omedelbart och krypteras. Personuppgifterna kan delas in om ytterligare uppgifter om de anställda registreras.

2.2 Manuell kodgranskning

Vid den manuella kodgranskningen har kod studerats för att få en bättre inblick i hur applikationen är strukturerad. Jag började med en mindre genomgång av all kod för att sedan gå in mer detaljerat på de områden i koden som behandlar känsliga uppgifter och data. Det är väsentligt att hålla koll på allt som applikationen tar emot och skickar ut (Andersson, 2019).

Det jag fann var att de input-parametrar, där användare matar in data, saknade validering och s.k. *Prepared Statements* när olika anrop till databasen användes. I koden såg jag även att användare kan ladda upp filer, men att dessa inte kontrolleras huruvida dem innehåller filnamn med specialtecken eller dylikt. Detta kan användas för att komma åt filer som användaren inte ska komma åt.

Även filformatet och dess innehåll kontrolleras inte och kan leda till att opålitliga filer laddas upp. All input-data ska genomgå validering vid dess ingångspunkt. Detta uppfylls inte av applikationen som den ser ut nu. Under inloggningssidan används en timeout på fem minuter som ställs in i webconfig-filen. Fem minuter är en väldigt kort tid som ändrats till att vara mindre än grundinställningen för ramverket.

Användning av egenskaps-attribut som *HttpOnly* och *HttpPost* kunde jag inte hitta i koden. Detta höjer säkerheten genom att se till att ändra striktheten på metoder och förhindrar att skript kan ta över cookies. För att motarbeta *Cross-site request forgery* bör *Anti-Forgery Tokens* implementeras dessutom.

2.3 Kodgranskning med verktyg

En kodgranskning med verktyget *FxCop*, som tillhör *Microsoft Visual studio*, genomfördes för att vidare undersöka säkerheten i applikationen. Verktyget är ett statiskt sådant som analyserar kompilerad objektkod, inte källkoden. Granskningar av denna sort har en fördel i att snabba lösningar ofta föreslås när brister har identifierats av verktyget (Microsoft, 2018). Om lösningar inte ges kan utvecklaren åtgärda det direkt i och med att felet visas direkt i utvecklarprogrammet (likt vanliga kompileringsfel).

Resultatet jag erhöll av denna analys var likt det jag själv observerat i den manuella granskningen. SQL-anrop till databasen behövde ses över då anropen använde sig av en inkommande string-parameter från användaren. Verktyget gav då tips om att använda *Stored Procedures*, alternativt att ändra så att anropen parametriseras, istället för att byggas upp med användares input.

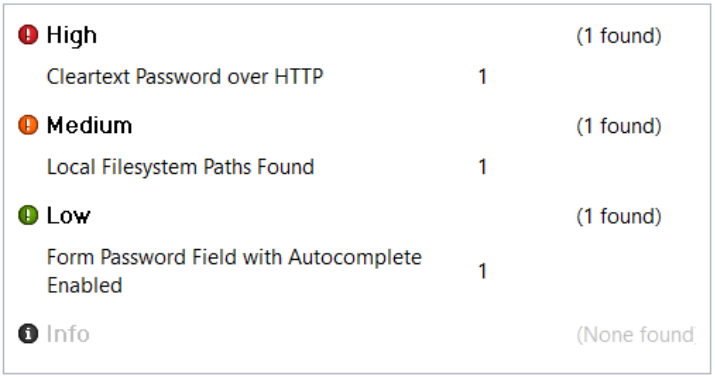
Detta poängterades ut vid tio olika platser i databas-klassen. *FxCop* loggar även detta i en fil som jag sedan såg över. Samtliga observationer markerades med nivån varning. En potentiell lösning för detta är att använda *Prepared Statements*, vars huvudfunktion är att göra texten till ett SQL-anrop när det skapas.

Fördelen med denna lösning är att i de flesta fall, skickas anropet direkt till databashanteraren där den kompileras vidare. Det förberedda anropsobjektet innehåller alltså inte bara ett SQL-anrop men ett anrop som redan är förkompilerat. Konsekvensen blir att databashanteraren kan köra anropet utan att kompilera det först.

2.4 Enkel penetrationstestning

Två penetrationstester har genomförts i syfte att utvärdera säkerheten för applikationen genom att simulera en attack mot den. Sådana tester leder ofta till förslag kring lösningar för att åtgärda de brister som identifieras. Ytterligare en fördel med penetrationstestning är att specifika sårbarheter testas, vilket andra granskningar har svårare med att uppmärksamma. Dessa faktorer är bl.a. vad som gör penetrationstestning till ett viktigt steg i programutvecklings livscykel (OWASP, 2014).

Det första testet gjordes med verktyget Vega som skannar genom applikationen och pekar ut säkerhetsbrister (Subgraph, 2014). Verktyget är lättanvänt och är speciellt bra på att hitta svagheter inom filhantering, SQL-injektioner och olika former av *Cross-site scripting (XSS)*. Tack före öppen källkod är Vega ett säkert redskap att använda sig av då dess uppbyggnad kan studeras och kritiseras av allmänheten (Diehl, 2016).



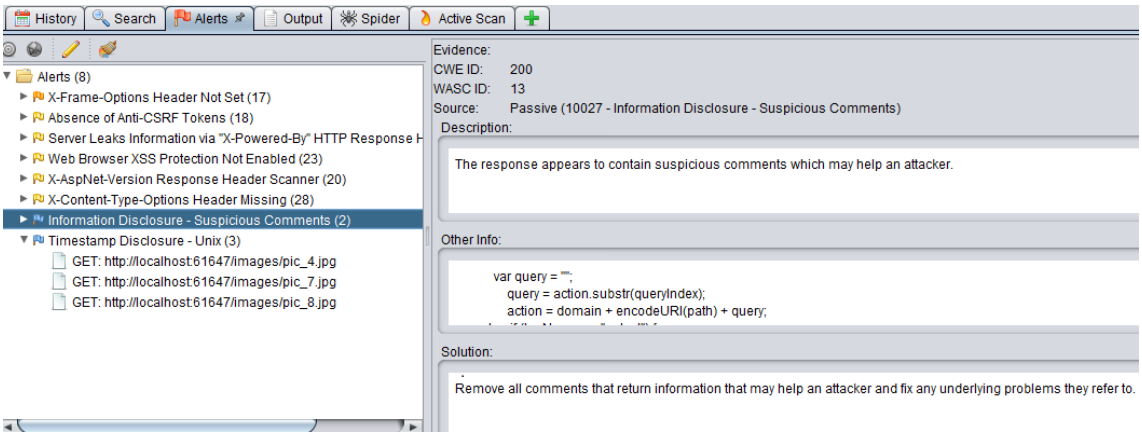
Figur 2

Resultatet av analysen (se figur 2) visade att den högsta säkerhetsbristen var att lösenord skickas över http i klar text. När en användare anger sitt lösenord, skickas det inte över en krypterad förbindelse. Detta är en stor säkerhetsrisk då det kan finnas någon/något som lyssnar på denna kommunikation och då har chans att se lösenordet i klar text. En sådan attack kallas för *Man-in-the-middle attack*, och kan förhindras genom att införa ett kryptobindande attribut (Microsoft, 2019).

Verktyget hittade även sökvägar till lokala filsystem som kan användas av attackerare i XSS-attacker och extern kodexekvering. Denna sårbarhet bedömdes som medium av Vega, men faktum är att den är med i OWASP lista över de tio mest kritiska sårbarheter hos webapplikationer (OWASP, 2014). För att förhindra dessa attacker bör filsökvägar lagras i en databas och aldrig visas öppet. Helst ska också filers innehåll sparas i databasen, men beroende på storlek är detta inte alltid en möjlighet.

Den sista sårbarheten, med lägst säkerhetsnivå, var att lösenordsfält har autoslutförande aktiverat. På så sätt kan användare spara ner inmatade lösenord lokalt på datorn, vilket innebär en säkerhetsrisk då dessa lokala filer kan kapas. Denna sårbarhet kan förhindras genom att lägga på attribut på de platser där inloggningsformulären skapas.

Det andra verktyget kallas ZAP och är just nu världens mest populära säkerhetsverktyg som aktivt genomgår uppdateringar av ett internationellt team (OWASP, 2020). Detta verktyg påminner om Vega i sin enkelhet och genom att följa samma princip om att ha öppen källkod som gör verktyget pålitligt och lämpligt för dess ändamål (Diehl, 2016).



Figur 3

ZAP identifierade och varnade om åtta stycken olika problem vid analysen. Det första var att *X-Frame-Options* headern inte var inställd i applikationen. Denna inställning talar om för enskilda webbläsare hur den ska behandla webapplikationen och hjälper till med att förhindra Clickjacking-attacker (OWASP, 2020). Den tredje varningen har att göra med samma inställning. Sårbarheterna behöver åtgärdas på webbservern för att det ska förhindras i dessa fall.

Den andra varningen handlar om avsaknaden av *Anti-CSRF* Tokens. CSRF utnyttjar pålitligheten mellan server och webbläsare för att manipulera sig till förtroende hos applikationen. Denna sårbarhet åtgärdas genom att lägga på en token i formulär som kontrollerar huruvida POST-requests stämmer överens med hur dem ska se ut (OWASP, 2020).

Varning fyra har redan påpekats i denna rapport då den handlar om XSS. Den noterades givetvis ändå av verktyget vilket påvisar applikationens behov att åtgärda detta problem omgående. De nästa två varningar har att göra med information om http-Headern. Informationen kan avslöja detaljer om ramverket som används, vilket ger användbar information till attackerare (OWASP, 2020).

De två sista varningarna har med informationen som visas att göra. Avslöjande information kan leda till ledtrådar om applikationens struktur och tidsstämplar är något som inte bör visas. Läckt information av denna typ kan utnyttjas genom social manipulation och för att hitta andra kryphål som attackerare kan ta sig genom (Diehl, 2016).

2.5 Rekommendationer

Mina rekommendationer för applikationen är följande:

- Öka tiden för timeout vid misslyckade inloggningsförsök till 15 minuter. Detta bör göras för att sätta mer stränga krav och förstärka skydd mot Brute-force attacker.

- Validera alltid data som tas emot av applikationen av användare genom att testa format, längd och typ. Begränsa storleken på filer som kan laddas upp för att minska risken för belastning som kan skada applikationen.

- Genomför en check av string-variabler och acceptera bara dessa om de stämmer överens med förväntade värden. Neka andra resultat samt de som innehåller binära data, kommentars-tecken och "Escape"-sekvenser (ex. omvänt snedstreck).

- Bygg aldrig upp SQL-anrop med hjälp av inkommande parametrar från användare. Använd istället *Stored Procedures* för att validera det data som tas emot.

- Implementera ytterligare lager av validering för att säkra mot kryphål som kan förekomma av första valideringslagret.

- Sammanlänka aldrig inkommande data som inte validerats. Detta är annars ett av de enklaste sätten för oönskade skript att hitta en väg in i applikationen.

- Se till att lösenord som skickas över http alltid krypteras så dem inte blir kapade.

- Skriv aldrig ut filsökvägar eller detaljer om applikationen öppet, lagra dessa istället i en databas och koppla sökvägarna med denna.

- Lägg på skydd för att förhindra autoslutförande av lösenord med hjälp av attributet autocomplete="off" i lösenords form-taggar.

```
<form>
  Subject: <input type="text" name="subject"/><br/>
  Content: <input type="text" name="content"/><br/>
  <input type="submit" value="Submit"/>
  <input type="hidden" name="token" value="R6B7hoBQd0wfG5Y6qOXHPNm4b9WKsTq6Vy
6Jssxb"/>
</form>
```

Figur 4

- Använd Tokens vid formulär så att förtroende mellan server och webbläsare inte kan manipuleras (se figur 4).

```
<system.web>
  <httpRuntime enableVersionHeader="false" />
</system.web>
```

Figur 5

- Läck aldrig information som kan avslöja ramverkets infrastruktur (se figur 5).

3 Reflektion

Efter att ha skrivit denna rapport där jag fått applicera mina kunskaper om IT-säkerhet i en Case-studie, har jag kommit underfund med hur utvärderaren spelar en stor roll i användningen av välkända tekniker och verktyg. Även om säkerhetsverktyg är oerhört smidiga och utvärderar enligt en viss struktur, så krävs det fortfarande en bakomliggande utvärderare som gör något med resultatet.

Den manuella kodgranskningen kändes blir väldigt beroende av utvärderarens expertis. Jag kände till att granskningsmetoden har sin fördel i att färre falsk-positiv- och falsk-negativa resultat uppkommer. Dock kände jag att metoden fortfarande kan få varierande resultat till följd av den mänskliga faktorn. Jag förstår samtidigt vikten av denna granskningsmetod då det är den enda som har en kontext-medvetenhet, vilket kan vara till stor fördel i utvärderingar.

Min grundexpertis för programmeringstekniken som använts vid skapandet av applikationen hade en brist i sig. Detta var att .Net-Framework skiljer sig från .Net-Core som jag arbetat med tidigare. Vid studerande av dess skillnader observerade jag tydligt hur en stor säkerhetsbrist låg i utvecklares missuppfattande av lösningar som använder sig av Object Relational Mapping (ORM).

Flertalet tar det för givet att programmeringstekniken är säker mot SQL-injektioner, vilket inte är fallet. Tekniken har en språkuppsättning kallad LINQ och tillåter också inhemsk SQL som inte helt är säkrat mot sådana injektioner. Det tycks vara essentiellt att inte förenkla olika typer av attacker, som enligt mig är mer av samlingsord, där ett visst mönster av attacker beskrivs.

Det kodmönster jag själv följt under den manuella kodgranskningen för SQL-injektioner är där string-queries använder sig av en inkommande parameter och försöker sedan exekvera dynamiskt skapade queries innehållandes data som kan vara opålitligt. Detta mönster fungerade väl i granskningen, men jag insåg senare vikten av att inte utesluta ramverk och andra teknikers påverkan på attacker.

En kategori som jag kan tycka är viktig att inkludera i säkerhetsutvärderingar likt denna är integritet. En sådan kategori skulle jag ge en definition likt: där förstörelse eller ändring av information inte ska kunna göras

av personer eller system. Detta är en kategori som är svårare att testa, men bör vara en faktor för utvärderaren att ha i åtanke under olika granskningar.

Vidare har jag insett hur manuella granskningar och verktyg komplementerar varandra. Jag anser att båda är ytterst viktiga att använda sig av för att en komplett utvärdering ska kunna genomföras. Även här är utvärderaren själv viktig för att sammankoppla de varierande resultat som uppstår. Utvärderaren behöver i slutändan välja om verktygs varningsnivåer stämmer överens baserat på given kontext. Ett exempel i denna rapport var att jag ansåg att Vega's varningsnivå, som sattes på medium, egentligen är av högre nivå.

Problem som identifieras med statiska verktyg kan resultera i oavgörbara situationer i värsta fall. Anledningen till detta mynnar ut i det som kallas för Rice's sats, som säger att varje signifikant fråga om ett program kan förenklas ner till stopproblemet. Den praktiska slutsatsen av detta är att statiska verktyg blir tvungna till att göra avrundningar av problem, vilket i sin tur kan leda till bristfälliga utfall.

Detta har varit en intressant uppgift att ta sig ann och ett gynnsamt sätt att lära sig mer om säkerheten för webbapplikationer. Efter att ha genomfört en utvärdering som denna, känner jag att det kommer bli enklare att tänka på säkerheten först, när jag själv utvecklar applikationer. Detta är den främsta insikten jag tar med mig till framtida uppgifter.

4 Referenslista

Diehl, E. (2016). *Ten Laws for Security*

MICROSOFT. 2018. *FxCop*. Tillgänglig: https://docs.microsoft.com/en-us/visualstudio/code-quality/install-fxcop-analyzers?view=vs-2019 [Hämtad 2020-01-19]

MICROSOFT. 2019. *Man In The Middle*. Tillgänglig: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-sstp/4a6778bc-a4a9-46c6-9120-7493c61f95e5 [Hämtad 2020-01-19]

SUBGRAPH. 2020. *Vega*. Tillgänglig: https://subgraph.com/vega/ [Hämtad 2020-01-19]

OWASP. 2014. *OWASP Testing Guide*. Tillgänglig: https://owasp.org/ [Hämtad 2020-01-19]

OWASP. 2020. *Code Review Guide*. Tillgänglig: https://www.owasp.org/index.php/OWASP_Code_Review_Guide_Table_of_Contents [Hämtad 2020-01-19]