Murphys onda tvilling och en hackers syn på bra design

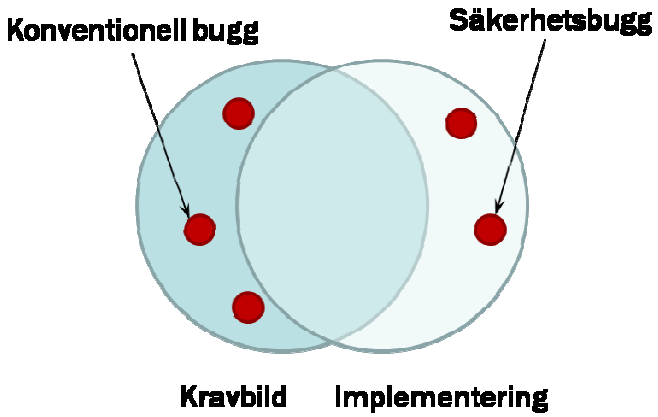
”Reliable software does what it is supposed to. Secure software does what it is supposed to, and nothing else.” –Ivan Arce

1 Tester

Inom programvaruutveckling, och ingenjörskonst generellt, genomförs ofta tester för att verifiera att resultatet har önskade egenskaper och fungerar korrekt. Man kan grovt skilja på två olika typer av tester; funktions- och säkerhetstester.

1.1 Funktionstester

Ett funktionstest kontrollerar att systemet har de egenskaper som är önskade (kravställda), att rätt funktioner har implementerats och att de fungerar som tänkt. Funktionstester hittar vad vi här kallar *konventionella buggar*; funktionalitet som saknas eller inte löser sin uppgift korrekt. Programmet sparar inte dokument automatiskt eller avrundar beräkningar fel. Konventionella buggar upptäcks ofta av att systemet bara används som det ska. Samtidigt är det inte svårt att tänka sig att en funktion i ett program stämmer överens med kravbilden och passerar ett funktionstest men samtidigt har buggar. De kanske bara dyker upp under särskilda förhållanden. Dessa kan ses som oönskad funktionalitet utöver kravbilden. Figur 1 beskriver skillnaden i form av ett venn-diagram. Den implementering som levereras (ljus cirkel) stämmer förstås aldrig helt överens med kravbilden (mörk cirkel), med konventionella buggar avses krav som inte är ordentligt uppfyllda. I fältet längst till höger, det som inte överlappar kravbilden, återfinns funktionalitet som inte finns i kravbilden. Där finns också *säkerhetsbuggar*.



Figur 1. Skillnaden mellan konventionella buggar och säkerhetsbuggar (Herbert H. Thompson & James A. Whittaker, 2002).

1.2 Säkerhetstester

En funktion i ett program kan fungera precis som specificerat och därmed passera ett funktionstest. Samma funktion skulle dock kunna misslyckas totalt i ett *säkerhetstest*. Anta till exempel att en inloggningsprocedur tar emot ett användarnamn/lösenords-par, jämför med vad som finns lagrat i databasen över registrerade användare, nekar inloggning om det inte stämmer men accepterar om paret är känt. Funktionstestet skulle passeras utan problem, korrekt användarnamn och lösenord ger tillgång men felaktiga nekas med felmeddelande. Anta samtidigt att alla inloggningsförsök där det

angivna lösenordet är längre än 64 tecken godkänns förutsatt att användarnamnet är korrekt. Detta skulle innebära att en angripare skulle kunna bryta sig in i valfritt konto, så länge användarnamnet är känt, endast genom att ange 65 godtyckliga tecken som lösenord. (Bortse för ögonblicket hur det kommer sig att funktionen beter sig på detta vis.) Detta innebär ett kolossalt säkerhetshål som inte bryter mot kravspecifikationen och är något som sällan upptäcks av misstag eller genom vanlig användning. Säkerhetstester syftar till att hitta sådana buggar.

2 Murphy och Jeff

Om det finns fler sätt att göra något och ett av dessa sätt leder till en katastrof så kommer någon att göra på det sättet. Alla känner till Murphys lag och den är aktuell för alla som tillverkar system som ska användas av människor. Mindre känt är dock att Murphy har en ond tvilling. De är båda duktiga på att hitta buggar.

2.1 Murphy

Murphy gör ofta precis det du inte hade tänkt eller hoppats på. Oftast handlar det om slarv och oskyldiga missuppfattningar; indata anges i fel format, procedurer genomförs i fel ordning och nödvändiga aktiviteter utelämnas. Om lösenordsautentiseringen som nämns i exemplet ovan gjorde skillnad på versaler och gemener skulle Murphy se till att caps lock trycks in av misstag då och då så att ”rätt” lösenord inte fungerar. Supportsamtalet ligger aldrig långt bort när Murphy är i farten.

Murphy är bra på att upptäcka bristande funktionalitet, saker som inte fungerar som de ska och dåliga gränssnitt. Saker som tenderar att göra kunder missnöjda. Dessvärre hittar Murphy sällan säkerhetsbuggar.

2.2 Jeff

Vad många inte vet är att Murphy har en elak tvilling, låt oss kalla honom Jeff. Jeff skiljer sig från sin bror på många sätt; istället för klumpig är han beräknande. Istället för omedveten är han högst eftertänksam. Istället för att göra väsen av sig arbetar han hårt för att inte märkas. Jeff är grannen som verkar reko men samtidigt ger dig en känsla av att något är fel.

Jeffs mål är att hitta detaljer som det i exemplet med långa lösenord. Jeff är uteslutande ute efter säkerhetsbuggarna och han söker aktivt.

Vad som gör Jeff farlig är att han är effektiv, systematisk, vet precis vad som påverkar olika system och förstår vad som får dem att fallera. Jeff hittar funktionalitet och fel som sällan upptäcks av misstag och som användare inte lider av under vardaglig användning. Samtidigt finns för honom ett stort övertag i att Murphys problem tar en stor del av utvecklarnas tid så att han kan arbeta ostört.

För att stoppa Jeff behövs radikalt annorlunda åtgärder jämfört med Murphy.

3 Säker design

Punktinsatser, att genom återkommande säkerhetstester hitta och fixa säkerhetsbuggar, är generellt dyrt och ineffektivt. I vanlig ordning är det mest förmånligt att tänka efter före, är systemet redan utvecklat har man dock få val. För att underlätta säkerhetsarbetet redan under planering och design kan man använda sig av ett antal grundläggande principer som beskriver hur säkra system bör byggas.

Här presenteras sju stycken erkända säkerhetsprinciper, samtliga är hämtade från välkänd litteratur på området (*Secure Programming for Linux and Unix HOWTO* av David A. Wheeler samt *Writing Secure Code* av Michael Howard & Steve Lipner).

3.1 Minimize attack surface

Begreppet attackyta används tyvärr inte i särskilt stor utsträckning i säkerhetsbranschen och än mindre utanför den. Attackyta kan löst ses som mängden möjligheter en angripare har att interagera med ett system. Det är inte ett exakt mått utan används huvudsakligen för att grovt jämföra olika system eller samma system i olika konfigurationer. För programvara kan det exempelvis handla om konkreta mätetal som antal nätverksportar som lyssnar, antal filer som data hämtas från eller, mer abstrakt, mängden programkod som kan nås med fjärranrop. För en dörr till en fastighet kanske det istället handlar om antal och typ av lås, utrymme för bräckjärn, synliga gångjärn, portkod, kattlucka, dörrfönster, brevinkast, etc.

Principen är i grunden enkel, ju mindre attackytan är desto mindre är sannolikheten att säkerhetshål kan hittas och utnyttjas av utomstående. För applikationer innebär det bland annat att inte implementera komplicerad funktionalitet i de fall det inte behövs och att begränsa åtkomst till sådan i andra fall.

3.2 Don’t mix code and data

En dokumentfil är data; det kan vara text, bilder, ljud, film eller dylikt. Kod är något annat, det är instruktioner som en processor (eller program) ska genomföra. Det klassiska exemplet är exekverbara filer med maskinkod som till exempel exe-filer. Utöver detta inkluderas även skriptspråk som Perl, Python och VBScript.

Data är data och kod är kod, utom när det inte är det. Många gånger genom historien har dessa två fenomen blandats så att kod smygs in i data. Exempel på detta är makron i textdokument, Javascript i webbsidor och den grundläggande arkitekturen i de processorer vi använder i persondatorer. Dessa förekomster har utnyttjats för makrovirus i ordbehandlare, cross-site scripting (XSS) på webben och buffer overflows i kompilerade program. Flera andra attacker, till exempel SQL injection, bygger i grunden på samma problem.

Rå data i information kan inte skada någon på samma sätt som instruktioner i ett program kan. Ett fotografi ska inte kunna leda till tillståndsförändringar. Så fort detta sker, att fotot kan utföra handlingar, kommer det att utnyttjas. Kod och data ska blandas med stor försiktighet.

3.3 Security features != secure features

Med ”security features” avses funktionalitet som uteslutande handlar om säkerhet och inte fyller ett eget syfte i ett systems användning. Det handlar bland annat om inloggning, kryptering, åtkomstkontroll, behörighetsnivåer, indatavalidering och så vidare. Principen kan tolkas på två sätt och båda är relevanta. Dels som att (1) säkerhetsfunktionalitet inte betyder att systemet är säkert. Internetbanken är inte säker bara för att kommunikationen mellan bank och dator är krypterad, även om krypteringen är riktigt utförd. Vidare kan principen uppfattas som att (2) säkerhetsfunktioner inte behöver vara säkra, de kan ha buggar. Återigen är exemplet ovan med lösenord på över 64 tecken relevant. Tolkningarna överlappar varandra men båda lyder under samma krav. Säkerhetsfunktionalitet är alltid kritisk, svår att konstruera korrekt och måste granskas noggrant.

3.4 Fail safe

Fail-safe är en gammal ingenjörsprincip som är relevant för allt från bilar, tåg och flyg till kärnkraftverk och datorsystem. Om det värsta inte kan förhindras och inträffar ska skadorna åtminstone minimeras. Ett väldigt förklarande exempel ges av:

```
int ret = IsAccessAllowed(...);
if (ret == ERR_ACCESS_DENIED) {
  // inform that access is denied
}
else {
  // security check okay
}
```

Vad händer om IsAccessAllowed() misslyckas på grund av någon extern anledning som att till exempel minnet tar slut? Om funktionen returnerar ERR_OUT_OF_MEMORY faller if-satsen igenom till else och autentiseringen ”lyckas”. Tänk till exempel en dörrvakt på en nattklubb som släpper in personer med legitimationer som inte går att tyda. Om legitimationen säger att du är för ung får du gå hem men om du är tillräckligt gammal *eller* det inte går att avgöra vad det står på kortet, kommer du in. Låter det bekant från exemplet i avsnitt 1 ovan om inloggningen som alltid godkänner lösenord med fler än 64 tecken?

Logiken måste skrivas så att systemet hamnar i ett säkert tillstånd när någonting oförutsett händer. Ibland talas det om ”default to denial of service” när det gäller säkerhetsimplementeringar. Jämför med ett kassaskåp med elektroniskt lås som blir fuktigt och kortsluter. Vad är bäst, att skåpet automatiskt låses permanent eller att det öppnas? Naturligtvis beror det på vad det är för system då samma sak naturligtvis inte gäller för dörrlåset till brandtrappan.

3.5 Minimize feedback

Varje uns av information som ges till en angripare tas emot med öppna armar. Det kan röra sig om vad som helst, från uppenbart användbar information som administratörens lösenord till vilken tidszon servern befinner sig i. För en kompetent angripare kan även, till synes, obetydliga underrättelser vara av nytta.

Generellt talar man om två olika former av feedback som vill undvikas. (1) Beskrivande felmeddelanden och (2) informationsläckage. Ett klassiskt exempel på den tidigare är den stack trace som visas för webbesökare när .NET eller Java råkar ut för ett undantag. Besökaren får direkt reda på vilka bibliotek som används och kan börja göra antaganden om programmets logik. Informationsläckage är ofta mer subtilt, det kan röra sig om en inloggningsfunktion som beter sig något annorlunda om det är användarnamnet som är fel jämfört med ett felaktigt lösenord. I sådana fall kan en angripare systematiskt först gissa sig till korrekta användarnamn för att därefter testa vanliga lösenord mot dessa konton.

3.6 Use least privilege

Moderna persondatorer använder sig på processornivå av två säkerhetsnivåer; operativsystem och programvara. Den tidigare åtnjuter privilegier som den senare inte har. Nivåskillnaden syftar till att förhindra användares program att av misstag eller avsiktligen sabotera operativsystemet. Ovanpå

detta, i programmen, finns liknande konstruktioner; administratörer och vanliga användare har olika behörighet till kod och data. Behörighetsnivåer är en grundsten inom datorsäkerhet och återfinns ofta i filsystem, databaser och i det mesta som hanterar data.

En process (ett program under körning) har från början samma privilegier som den användare som startade den, processens *ägare*. Detta innebär att ett program med en sårbarhet som kan utnyttjas för att utföra någon oönskad handling, genomför denna med dess ägares rättigheter. Således är sårbarheter i högpriviligierade program, generellt, en större säkerhetsrisk än sådana i program med begränsade rättigheter. Detta är en av anledningarna till att användare ombeds att inte använda administratörskontot till vardags för surfning och mail. En angripare som tar över mailklienten får då administratörsrättigheter.

Principen att följa är att som programmerare inte ta sig större rättigheter än vad som krävs. Om särskilda friheter behöver tas under uppstarten av ett program ska så många av dessa som möjligt släppas så tidigt som möjligt, permanent. Om ett program behöver öppna TCP-portar mellan 1 och 1024 så kräver detta root-rättigheter. Så fort porten är öppnad behövs inte dessa rättigheter längre utan kan kastas. Eventuella angripare som utnyttjar sårbarheter och får kontroll över programmet kan sedan inte orsaka lika stor skada.

3.7 Never trust external systems

Det har länge varit accepterat att en server måste vara på sin vakt. Genom tiderna har vi varit med om buffer overflows i inloggningsprocedurer, SQL injection i webbapplikationer och format string-attacker i FTP-kommandon. Detta är alla exempel på attacker som en server utsätts för via sitt gränssnitt mot klienten. Det är inte alltid så enkelt. Det finns mängder av exempel på ordbehandlare som måste vara försiktiga när de öppnar dokument, FTP-klienter som inte kan lita på servern de kopplar upp sig mot och vanliga arbetsstationer som inte kan dela filer med likasinnade utan risk för attack.

Begreppet attackyta har redan diskuterats ovan i avsnitt 3.1, applikationers attackyta sträcker sig ofta vida utanför klientens gränssnitt. Kod och data hämtas bland annat från lokala filer, databaser och Windows register. Trenden med Web 2.0 och dess webbtjänster och mashups verkar göra situationen värre. Kan man förvissa sig om att det där RSS-flödet verkligen levererar det man förväntar sig och hur används de data som tas emot?

En robust applikation måste kunna hantera att externa system inte är pålitliga. Till vilken grad detta ska kunna hanteras varierar förstås beroende på applikationen själv och systemet i stort. Generellt kan det sägas att ju mindre man behöver lita på externa system desto mer robust är applikationen.

4 Avslutning

Murphy har länge varit ett aber för systemutvecklare, men Jeff förtjänar också att uppmärksammas. En anledning till att Jeff hamnat i skymundan är att Murphys lag postulerar att katastrofen verkligen kommer att inträffa om det är möjligt. Det är inte lika säkert att Jeff dyker upp. Många gånger då Jeff har dykt upp har det inte alltid upptäcks förrän långt senare. Jeff representerar det plötsliga, det oväntade. Jeff ger oväntade utgifter som det inte budgeterats för på samma sätt som Murphy. Och kostnaden då Jeff väl dyker upp kan vara väldigt hög. Det kan därför vara bättre att ta en stadig men lägre utgift som man budgeterar för än att plötsligt drabbas av en stor icke-budgeterad utgift. Få skulle drömma om att gå utan hemförsäkring trots att olyckan inte alltid inträffar.