Biztonsági API analízis a spi-kalkulussal

Buttyán Levente ***Ta Vinh Thong***
buttyan@crysys.hu ***thong@crysys.hu***

CrySyS Adatbiztonsági Laboratórium
BME Híradástechnikai Tanszék

Kivonat: *Az API szintű támadások komoly veszélyt jelentenek a hardver biztonsági modulokra nézve, ezért fontos követelmény az API-ban rejlő biztonsági lyukak felfedezése és foltozása. Az API analízis egyik igéretes iránya a formális verifikációs módszerek alkalmazása. Cikkünkben ezt az irányt követjük, s egy processz-algebra alapú API verifikációs módszert javaslunk, mely különösen alkalmasnak látszik a biztonsági API-k működésének formális leírására, a biztonsági követelmények precíz definiálására, és a megfogalmazott követelmények teljesítésének ellenőrzésére. Munkánk motiválása céljából ismertetünk néhány konkrét API szintű támadást is egy a gyakorlatban elterjedten használt hardver biztonsági modul ellen.*

Bevezetés

Számos alkalmazásban használnak hardver biztonsági modulokat (Hardware Security Module, vagy röviden HSM). HSM alatt olyan hardver eszközt értünk (a rajta futó förmver és szoftver komponensekkel együtt), mely bontás-ellenálló (tamper resistant) tulajdonságokkal rendelkezik, s ezáltal alkalmas kriptográfiai kulcsok biztonságos tárolására, valamint különböző biztonság-kritikus kriptográfiai algoritmusok (pl. digitális aláírás generálás, PIN-kód generálás) végrehajtására.

A hardver biztonsági modulok polgári célú alkalmazása a bankszférában kezdődött az 1960-as években. Az ebben az időszakban történő bankkártya hamisítások arra ösztönözték az IBM-et (mint a kor banki számítástechnikai rendszereinek legfőbb szállítóját), hogy kifejlesszen egy olyan rendszert, amely lehetővé teszi a felhasználók PIN-kódjának előállítását a bankkártyán tárolt számlaszámból egy PIN-kód származtatási kulcs (PIN derivation key) segítségével. Ennek kapcsán szükségessé vált a PIN-kód származtatási kulcsok megfelelő védelme mind külső támadók mind pedig a bank belső alkalmazottai ellen. Ez a követelmény vezetett az IBM 3848, első generációs HSM kifejlesztéséhez, melyet később széleskörben alkalmaztak a banki ATM hálózatokban. Mára a HSM-ek alkalmazási köre kiszélesedett, s a banki alkalmazásokon túl, elterjedten használják őket például a nyílvános kulcs infrastruktúrákban (Public Key Infrastructure, vagy PKI), a tömegközlekedési elektronikus díjbeszedési rendszerekben, és általában az elektronikus kereskedelem területén.

A HSM-ek elleni klasszikus támadási módszerek a fizikai támadások **[2]**. Ezek lehetnek a hardver modul fizikai megbontásával, esetleg roncsolásával járó intrúzív támadások, vagy a HSM működési környezetének (pl. időzítéseinek, áramfelvételének) megfigyeléséből és manipulálásából származó támadások. A fizikai támadások hatékonyak, ám sokszor költséges berendézeseket igényelnek.

A fizikai támadások mellett, a közelmúltban megjelentek a jóval kisebb költséggel járó szoftver alapú támadások, melyek a HSM alkalmazás programozói interfészében (Application Programming Interface, vagy röviden API) rejlő gyengeségeket, hibákat aknázzák ki. Számos elterjedten használt (s különben erős fizikai védelmet biztosító) HSM ellen találtak API szintű támadást **[3, 4, 5, 6, 7, 10, 11]**. Nyílvánvaló, hogy kívánatos lenne az API-ban rejlő biztonsági lyukak felfedezése és foltozása, ideálisan még az adott HSM széleskörű telepítése előtt.

Ugyanakkor, a gyakorlatban használt API-k több száz függvényt tartalmazó komplex rendszerek, ami megnehezíti az analízisüket.

Az API analízis egyik igéretes iránya a szoftverfejlesztés területén használt formális verifikációs módszerek alkalmazása **[8, 9, 11, 12, 14, 15]**. Cikkünkben ezt az irányt követjük, s egy processz-algebra alapú API verifikációs módszert javaslunk, mely különösen alkalmasnak látszik a biztonsági API-k működésének formális leírására és a biztonsági követelmények precíz definiálására. Konkrétan, az itt bemutatott módszer a spi-kalkulusra épül **[1]**, melyet eredetileg kulcscsere protokollok analízisére fejlesztettek ki. Ismereteink szerint mi használtuk először a spi-kalkulust biztonsági API-k analízisére.

A továbbiakban először egy konkrét HSM (a Visa Biztonsági Modul) elleni API szintű támadásokat mutatunk be illusztratív céllal. Hasonló támadások léteznek más HSM-ek ellen is. Ezek a támadások motiválják a 4. fejezetben bemutatásra kerülő API analízis módszert, melynek alapját a 3. fejezetben ismertetésre kerülő spi-kalkulus képezi.

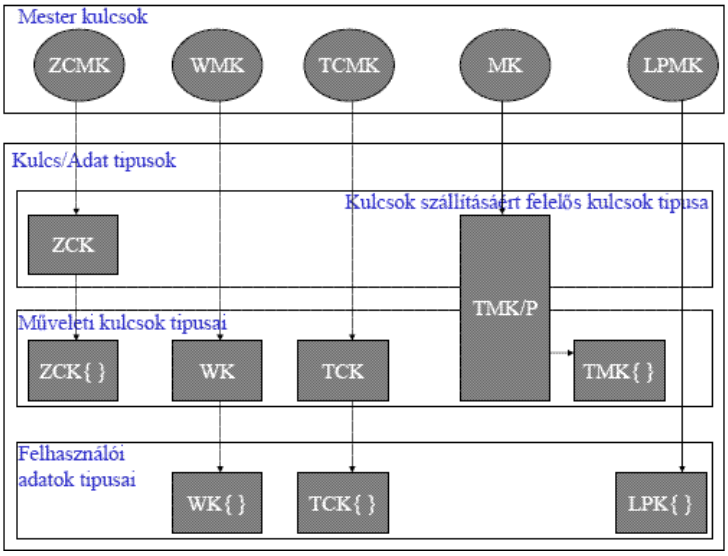
A Visa Biztonsági Modul támadása az API-n keresztül

A Visa Biztonsági Modul (Visa Security Module, vagy VSM) kifejlesztésével a Visa célja az volt, hogy meggyőzze a hozzá tartozó tagbankokat, hogy csatlakoztassák ATM-jeiket a Visa hálózatához, és ezáltal lehetővé váljon, hogy bármely tagbank ügyfele pénzt vehessen fel egy olyan ATM-ből, amely egy másik tagbankhoz tartozik. Ennek érdekében a Visa-nak biztosítania kellett, hogy bármely tagbank más tagbank gondatlanságából származó esetleges vesztesége a lehető legkisebb legyen. Ez többek között azt is jelenti, hogy az egyes tagbankok ügyfeleinek PIN-kódját, más tagbankok belső alkalmazottai nem tudhatják meg. Azaz a PIN-kódokat nem lehet egyszerűen a bankok mainframe-jein futó szoftverben kezelni. Ezért a PIN-kódok kezelése a fizikai védelmet biztosító VSM-ekben történik.

Mivel a HSM-ek belső tárkapacitása korlátos, ezért általában csak a legfontosabb kulcsokat (az ú.n. mesterkulcsokat) tárolják a HSM-ben. Minden más kulcsot a típusuknak megfelelő mesterkulccsal kódolják, és külső tárban tárolják. A megszokott tárolási mód a hierarchikus struktúra, amelynek előnye, hogy hatékony és áttekinthető. Hátránya azonban, hogy ha egy felsőszintű kulcs kompromittálódik, akkor minden a hierarchában alatta elhelyezkedő kulcs is kompromittálódik.

A VSM kulcshierarchiája az 1. ábrán látható. A VSM kilenc kulcstípust támogat, ezeket az ábrán a téglalapok jelképezik. A kulcshierarchia legfelső szintjén helyezkednek el a mesterkulcsok, melyek a VSM-en belül tárolódnak. Minden más kulcsot ezekkel a mesterkulcsokkal kódolva külső tárban tárolnak. Látható, hogy a belül tárolt mesterkulcsokból öt darab van. A *ZCMK* az a mesterkulcs, amivel az összes *ZCK* típusú kulcsot kódolják. A *ZCK* típus a zóna vezérlő kulcsokat (Zone Control Key) jelöli. Ezek a kulcsok a különböző bankhálozatok között vannak megosztva, és a bankhálozatok közötti kulcscserében játszanak szerepet. A *WMK* az a mesterkulcs, amivel az összes *WK* típusú kulcsot kódolják, ahol a *WK* a munka kulcsokat (Working Key) jelöli. A *WK* típusú kulcsok funkciója az, hogy a beütött PIN-kódot védjék miközben az eljut a banki hálozaton keresztül ahhoz a bankhoz ahol ellenőrizni tudják. A *TCMK* mesterkulccsal kódolják a *TCK* típusú kulcsokat, ahol a *TCK* típus a terminál kommuninkációs kulcsokat (Terminal Communication Key) tartalmazza. A terminál kommunkációs kulcsok funkcióihoz tartozik a VSM-ek között cserélendő üzenetek integritásvédő kódjának kiszámítása. Az *MK* mesterkulcs a *TMK* terminál-mesterkulcsok és a *P* PIN-kód származtatási kulcsok kódolásáért felelős. A *TMK* kulcsot később még tágyaljuk. Mivel a *TMK* kulcsokat más kulcsok kódolására használják (pl. a zóna kulcsok kódolására), a *P* kulcs pedig a PIN-kód kiszámításában játszik szerepet, ezért

helyezkedik el mindkét típus két hierarchia szinten is. Mivel nem lényegesek jelen cikk szempontjából, ezért az *LPMK* és *LPK* kulcsokat nem tárgyaljuk. Az *X{}* típusba olyan kulcsok vagy adatok tartoznak, amelyeket az *X* típusú kulccsal kódoltak.



1. ábra. A Visa Biztonsági Modul (VSM) kulcshierarchiája

Egy új ATM üzembehelyezésekor, a banknak el kell juttatnia az új ATM-nek a működéshez szükséges kulcsokat. Ehhez először a bank egy új terminál-mesterkulcsot (*TMK*) oszt meg az ATM-mel, majd minden más kulcsot ezzel a *TMK* kulccsal kódolva juttat el az ATM-hez. A *TMK* kulcs létrehozása a következő módon történik. A hoszt meghívja a VSM API-jának *GenerateKeyShares* nevű[1] függvényét:

$$Hoszt \rightarrow VSM : "GenerateKeyShares"$$

Erre a VSM generál egy TMK_i részkulcsot, majd egyrészt kinyomtatja a generált részulcsot a megfelelő biztonságos printeren:

$$VSM \rightarrow S\,\mathrm{ec}\,urePrinter : TMK_i$$

másrészt visszaadja a részkulcsot egy a VSM belsejében tárolt *MK* mesterkulccsal kódolva a hosztnak:

$$VSM \rightarrow Hoszt : \{TMK_i\}_{MK}$$

A hoszt annyiszor hajtja végre a fenti hívást amennyi részkulcsot szeretne generálni. A továbbiakban feltesszük, hogy a szükséges részkulcsok száma kettő.

A biztonságos printereken kinyomtatott részkulcsokat a meghatalmazott személyek eljuttatják az új ATM-hez. Miután megkapta mindegyik részkulcsot, az ATM előállítja a *TMK* terminál-mesterkulcsot a részkulcsok XOR-olásával: $TMK = TMK_1 \oplus TMK_2$. A banknál ugyanez a *TMK* kulcs áll elő a VSM API *CombineKeyShares* függvényének meghívása után:

[1] A cikkben használt függvénynevek nem mindenhol egyeznek a VSM specifikációban szereplő függvénynevekkel.

$$Hoszt \rightarrow VSM : "CombineKeyShares", \{TMK_1\}_{MK}, \{TMK_2\}_{MK}$$
$$VSM \rightarrow Hoszt : \{TMK_1 \oplus TMK_2\}_{MK} = \{TMK\}_{MK}$$

A fenti terminál-mesterkulcs generálási eljárás egy támadási lehetőséget rejt magában. Nevezetesen, a hoszt (illetve az azt kezelő alkalmazott) meghívhatja a *CombineKeyShares* függvényt két azonos paraméterrel (pl. a kódolt részkulcsok egyikével):

$$Hoszt \rightarrow VSM : "CombineKeyShares", \{TMK_1\}_{MK}, \{TMK_1\}_{MK}$$
$$VSM \rightarrow Hoszt : \{TMK_1 \oplus TMK_1\}_{MK} = \{0\}_{MK}$$

Az így létrehozott *TMK* a csupa nulla bitből alló kulcs. A VSM ezt a kulcsot használja többek között a *P* PIN-kód származtatási kulcs kódolására mielőtt az átküldésre kerül az ATM-nek. A VSM által előállított $\{P\}_0$ kódolt kulcsot azonban a támadó is könnyen dekódolni tudja a csupa nulla kulccsal. A *P* kulcs segítségével ezek után tetszőleges számlaszámhoz tartozó PIN kódot elő tud állítani a támadó.

Egy másik támadási lehetőség abból adódik, hogy a VSM API-ja tartalmaz egy *EncryptCommsKey* függvényt, amely egy *TCK* típusú kulcsot vár paraméterként és válaszként az ehhez a kulcstípushoz tartozó *TCMK* mesterkulccsal kódolva adja vissza a kulcsot:

$$Hoszt \rightarrow VSM : "EncryptCommsKey", TCK$$
$$VSM \rightarrow Hoszt : \{TCK\}_{TCMK}$$

Említettük, hogy minden szükséges kulcsot el kell juttatni az új ATM-hez, s ezalól a *TCK* kulcs sem kivétel. A szállítás a *TMK* kulccsal kódolva történik, így szükség van egy $\{TCK\}_{TMK}$ kulcstokenre. Ennek létrehozását a *TranslateCommsKeytoTMK* függvény biztosítja:

$$Hoszt \rightarrow VSM : "TranslateCommsKeytoTMK", \{TCK\}_{TCMK}, \{TMK\}_{MK}$$
$$VSM \rightarrow Hoszt : \{TCK\}_{TMK}$$

A támadás azt használja ki, hogy a VSM azonos *MK* kulcs alatt tárolja a *TMK* és a *P* kulcsokat. A támadás menete a következő:A bank rosszindulatú alkalmazottja kiadja az *EncryptCommsKey* parancsot, de paraméterként nem egy *TCK* kulcsot ad meg, hanem egy *PAN* felhasználói számlaszámot:

$$Hoszt \rightarrow VSM : "EncryptCommsKey", PAN$$
$$VSM \rightarrow Hoszt : \{PAN\}_{TCMK}$$

Ezután, a támadó meghívja a *TranslateCommsKeytoTMK* függvényt az előző lépésben kapott $\{PAN\}_{TCMK}$ értékkel és egy korábban megszerzett $\{P\}_{MK}$ rejtjelezett PIN-kód származtatási kulccsal mint paraméterrel[2]:

$$Hoszt \rightarrow VSM : "TranslateCommsKeytoTMK", \{PAN\}_{TCMK}, \{P\}_{MK}$$
$$VSM \rightarrow Hoszt : \{PAN\}_P = PIN$$

[2] Ez lehetséges mert *TMK* és *P* ugyanúgy az *MK* kulccsal vannak kódolva és a VSM csak azt nézi, hogy az adott kulcstoken sikeresen kódolható-e az *MK* kulccsal.

A visszakapott érték a PIN-kód származtatási kulccsal rejtjelezett számlaszám, azaz pontosan számlatulajdonos PIN-kódja.

A spi-kalkulus áttekintése

Ebben a fejezetben rövid áttekintést adunk a spi-kalkulusról **[1]**, ami a π-kalkulus **[13]** kiterjesztése különböző kriptográfiai primitívekkel. A π-kalkulushoz hasonlóan, a spi-kalkulus is egy egyszerű programozási nyelvnek tekinthető. Ennélfogva a spi-kalkulus kiválóan alkalmas a biztonsági API-k működésének modellezésére.

A spi-kalkulus nyelvtana

A spi-kalkulusban a kommunikációs csatornákat nevekkel jelöljük. Végtelen névhalmazt feltételezünk. Ezenkívül bevezetjük a változók végtelen halmazát is, amelyeknek az értékadásnál lesz majd szerepük. A változókat az *x*, *y*, és *z* betűkkel jelöljük, a neveket pedig többek között *m*, *n*, és *c* betűkkel. Két alapvető nyelvi elemet különböztetünk meg: *term*-eket (üzenetek, csatorna azonosítok, kulcsok, stb.), amik adatot reprezentálnak és *processz*-eket, melyek a viselkedést írják le. A termek lehetnek atomiak, mint a konstansok és változók, vagy összetett termek.

A termeket a következő nyelvtan szerint definiáljuk:

$L, M, N ::=$	*termek*
n	*név*
(M, N)	*pár*
0	*nulla*
$suc(M)$	*következő*
x	*változó*
$\{M_1, M_2, ..., M_k\}_N$	*szimmetrikus kulcsú titkosítás*

Tehát egy term lehet egy név, egy term pár, nulla, egy adott term utáni term, vagy egy változó. Külön kiemeljük továbbá az $\{M_1, M_2, ..., M_k\}_N$ formájú termeket, melyek szimmetrikus kulcsú titkosítással előállított kritpogramokat reprezentálnak, ahol *N* jelöli a kulcsot, az $M_1, M_2, ..., M_k$ termek pedig a nyílt üzenet mezőit.

A processzeket a következő nyelvtan szerint definiáljuk:

$P, R, Q ::=$	*processzek*
$\overline{M}\langle N_1, N_2, \text{K}, N_k\rangle.P$	*küldés* $(k \geq 0)$
$M(x_1, x_2, \text{K}, x_k).P$	*vétel* $(k \geq 0)$
$P \mid Q$	*(párhuzamos) kompozició*
$(vn)P$	*megkötés*
$!P$	*replikáció*
$[M \text{ is } N]P$	*összehasonlítás*
0	*null processz*
$let\ (x, y) = M\ in\ P$	*pár szétválasztás*
$case\ M\ of\ 0 : P\ suc(x) : Q$	*egész szám eset*
$case\ L\ of\ \{x_1, x_2, \text{K}, x_k\}_N\ in\ P$	*szimmetrikus kulcsú dekódolás* $(k \geq 0)$

Az egyes konstrukciók jelentése a következő:

- ***Küldés*** : Az M term itt egy csatornát reprezentál. Ez a processz kész a $N_1, N_2, ..., N_k$ termeket elküldeni az *M* csatornán keresztül. Ha egy üzenetváltás (lásd később) létrejön, akkor $N_1, N_2, ..., N_k$ elküldésre kerül az *M* csatornán keresztül és a P processz fut tovább.

- ***Vétel*** : Ez a processz az előző párja. Egy üzenetváltás során a *küldés* processz elküldi az $N_1, N_2, ..., N_k$ termeket mint üzeneteket az *M* csatornán, a *vétel* processz pedig ugyanezen a csatornán veszi ezeket a termeket és a $P\left[N_1 / x_1, N_2 / x_2, \text{K}, N_k / x_k\right]$ processz fut tovább, ahol N / x az értékadást jelöli. Azaz vétel során a vett termekkel mint értékekkel helyettesítjük a megfelelő változókat a *P* processzben.

- ***Kompozíció*** (P|Q): Ez a konstrukció a *P* és *Q* processzek párhuzamos futását jelöli. P és Q kommunikálhat egymással egy közös megosztott csatornán keresztül, vagy P és Q egymástól függetlenül kommunikálhat a környezettel.

- ***Megkötés*** $(vn)P$: A P processz létrehoz egy új n lokális nevet. A *P* processzen kívül más processzben – hacsak nem kapta meg explicite valamilyen kommunikáció során – ez a név nem jelenhet meg. E konstrukció segítségével modellezhetjük egy új titkos kulcs létrehozását.

- ***Replikáció*** $(!P)$: Ez a konstrukció a P processz végtelen sok példányának párhuzamos kompozícióját jelöli.

- ***Összehasonlítás*** $\left([M \text{ is } N]P\right)$: Ez a processz úgy viselkedik, hogy amennyiben $M = N$ akkor a P processz fut, különben a futás elakad.

- ***Null processz*** (0) : Ez konstrukció a semmitevést vagy elakadást jelöli.

- ***Pár szétválasztás*** $(let\ (x, y) = M\ in\ P)$: Ez a processz a termek nyelvtanában definiált párképzésnek az ellentettje. Ha $M = (N, L)$, akkor a $P[N / x][L / y]$ processz fut tovább, egyébként a futás elakad.

- ***Egész szám eset*** $(case\ M\ of\ 0 : P\ suc(x) : Q)$: Ez a processz úgy viselkedik mint P ha M értéke 0, vagy úgy mint $Q[N / x]$ ha $M = suc(N)$, különben elakad.

- ***Szimmetrikus kulcsú dekódolás*** : A $case\ L\ of\ \{x_1, x_2, \text{K}, x_k\}_N\ in\ P$ processz megpróbálja dekódolni az L termet az N kulccsal. Ha L egy $\{M_1, M_2, ..., M_k\}_N$ formájú term, akkor a $P\left[M_1 / x_1, M_2 / x_2, \text{K}, M_k / x_k\right]$ processz fut tovább. Különben a processz elakad.

A fent leírt, kriptográfiai elemeket használó nyelvi konstrukciók a következő alapfeltevésekre épülnek:

- Egy rejtjelezett üzenet csak a rejtjelezés kulcsának megfelelő dekódoló kulccsal fejthető meg.
- A rejtjelező kulcs nem következtethető ki a vele rejtjelezett üzenetből.

- A rejtjelezett üzenet elég redundanciát hordoz ahhoz, hogy a dekódoló algoritmus egyértelműen el tudja dönteni, hogy a dekódolás sikerült vagy nem.
- A támadó nem képes kitalálni és/vagy létrehozni bármilyen titkosnak minősített protokoll adatot.

A titkosság modellezése a spi-kalkulusban

A spi-kalkulusban a támadó egy tetszőleges R processz, melyről csak annyit tételezünk fel, hogy kezdetben nincsenek nála titkos adatok. A támadó processz párhuzamosan fut a rendszert modellező processzel, és azzal interakcióba léphet (kommunikálhat) a publikus csatornákat használva. Ezen interakció során szerzett információkból próbálja a támadó kinyerni a titkokat a rendszerből.

A *titkosság* mint biztonsági tulajdonság alapja a spi-kalkulusban a processzek *megkülönböztethetetlensége*. Azaz a protokoll titokban tart egy *M* adatot, ha tetszőleges *M'* adat esetén, a támadó *R* processz nem tud különbséget tenni a *P*(*M*) és a *P*(*M'*) processzek között, ahol *P*(*x*) a protokollt reprezentáló (paraméterezhető) processz.

A megkülönböztethetetlenség formális definiciója a *tesztelési ekvivalencia* fogalmára épül. Ennek megértéséhez be kell vezetnünk néhány további fogalmat:

- **Szabad és kötött változók:** A P processzben az x változó *kötött változó* ha P tartalmaz egy $m(x)$ vétel részprocesszt (tetszőleges m-re). A P processzben az x változó *szabad változó* ha P nem tartalmaz $m(x)$ vétel részprocesszt. Egy P processz szabad változóinak halmazát $fv(P)$-vel jelöljük.

- **Zárt processz :** Akkor mondjuk egy processzre, hogy zárt, ha nincs szabad változó benne. A spi-kalkulusban minden támadó processzről felteszük, hogy zárt.

- **Üzenetváltás :** Egy üzenetváltás akkor jön létre amikor egy $\overline{m}\langle M\rangle.P$ küldés processz és egy $m(x).Q$ vétel processz párhuzamos kompozicióban áll egymással. Ekkor a küldés processzes elküldi az M termet az m csatornán, ezt veszi a vétel processzt, és $P|Q[M/x]$ fut tovább. Formálisan

$$\overline{m}\langle M\rangle.P \mid m(x).Q \rightarrow P \mid Q\left[M / x\right]$$

- **Barb kimutatás:** A *barb kimutatás* intuitív jelentése, hogy egy processz használja-e az adott csatornát üzenet küldésre vagy fogadásra. A barb kimutatást a $\downarrow$ jelöli. A barb kimutatás teljességgel független a kiadott vagy kapott üzenettől. A barb kimutatásra a következő axiómák érvényesek:

    - **Barb In –** Ha egy processz *azonnal* használja az m csatornát adat fogadásra, akkor ő az m barb-ot kimutatja, azaz $m(x).P \downarrow m$**.**
    - **Barb Out –** Ha egy processz azonnal használja az m csatornát adat küldésre, akkor ő az $\overline{m}$ barb-ot kimutatja, azaz $\overline{m}\langle M\rangle.P \downarrow \overline{m}$.

- **Konvergencia**: A konvergencia intuitíven azt jelenti, hogy a processz nem feltétlenül azonnal használja az adott csatornát, hanem csak az üzenetváltásainak sorozata során valamikor használja azt. Ennek jelölése $\Downarrow$, és a kapcsolódó axiómák a következők:

- Ha egy processz a β barb-ot kimutatja, akkor konvergál a β-hoz.
- Ha egy P processz át tud alakulni egy olyan Q processzbe ami a β barb-ot kimutatja, akkor P konvergál a β barb-hoz.

Most, hogy a szükséges fogalmakat bevezettük, megadjuk a *tesztelési ekvivalencia* formális definicióját.

Definíció (Tesztelési ekvivalencia): Egy teszt egy (R, β) pár, ahol R egy tetszőleges zárt proccessz és β egy barb (m vagy $\overline{m}$). P és Q között fennáll a teszlelési ekvivalencia, azaz $P \approx Q$, akkor és csak akkor ha $P \subseteq Q$ és $Q \subseteq P$ egyszerre fennállnak, ahol $P \subseteq Q$ akkor és csak akkor áll fenn, ha minden (R, β) teszt esetén $(P|R)\Downarrow\beta$-ból következik $(Q|R)\Downarrow\beta$.

Intuitíven, $P \approx Q$ azt jelenti tehát, hogy a P és Q processzek megkülönböztethetetlenek egy külső R megfigyelő számára. Azaz, P és Q belső struktúrája lehet különböző, de ezt a P és Q-val párhuzamos kompozícióban levő harmadik zárt R processz nem tudja detektálni, a P-vel és Q-val folytatott üzenetváltások során.

Egy egyszerű API modellezése spi-kalkulussal

Bár a spi-kalkulust elsősorban kulcscsere protokollok modellezésére dolgozták ki, jól alkalmazható HSM-mel történő API-n keresztüli interakciók modellezésére is. Ez annak köszönhető, hogy egy API függvényhívás nagyon hasonló egy két lépéses protokollhoz, melyben a hoszt kiad egy kérést, és a HSM visszaad egy választ. A teljes API-t a lehetséges függvényhivásokat reprezentáló processzek párhuzamos kompozíciójával modellezhetjük. Erre mutatunk példát ebben a fejezetben.

Ennek érdekében először definiálunk egy egyszerű biztonsági API-t. Felteszük, hogy a HSM tartalmaz egy MK mesterkulcsot. Továbbá, megkülönböztetünk két kulcstípust, a K_i adatkódoló-kulcsot és a KEK_j kulcskódoló-kulcsot, amikhez hozzá rendeljük a $DataKey$ és $KEKKey$ típus indikátorokat. A K_i adatkódoló-kulcsot tartalmazó kulcstokenek $DataKey$ típus indikátort kapnak, míg a KEK_j kulcskódoló-kulcsot tartalmazó tokeneket $KEKKey$ indikátorokkal látjuk el. Továbbá bevezetünk egy $TData$ típusindikátort is, amit a felhasználói adatot tartalmazó rejtjeles szövegek típusának jelzésére használunk. Felteszük még, hogy a modul nem tárolja az adatkódoló-kulcsokat és a kulcskódoló-kulcsokat, helyette kiadja magából kulcstokenként ezeket $\{DataKey, K_i\}_{MK}$ és $\{KEKKey, KEK_j\}_{MK}$ formában.

Példa API-nk négy függvényt tartalmaz:

- **adat kódolás :** Ez a függvény argumentumként valamilyen *Data* felhasználói adatot és egy $\{DataKey, K_i\}_{MK}$ kulcstokent vár. Dekódolja a $\{DataKey, K_i\}_{MK}$-t az MK mesterkulccsal, ellenőrzi a kulcs típusát, és ha az $DataKey$**,** akkor K_i-vel kódolja a $Data$ adatot. Ezután visszaadja a $\{TData, Data\}_{K_i}$ rejtjelezett adatot.

- **adat dekódolás:** Ez a függvény argumentumként egy $\{TData, Data\}_{K_i}$ kódolt adatot és egy $\{DataKey, K_i\}_{MK}$ kulcstokent vár. Dekódolja a $\{DataKey, K_i\}_{MK}$-t az MK mesterkulccsal, majd ellenőrzi a kulcs típusát, és ha az $DataKey$ akkor K_i-vel

dekódolja a $\{TData, Data\}_{K_i}$-t. Végül ellenőrzi, hogy az típuss $TData$-e, s ha igen, akkor (és csak akkor) visszaadja a $Data$ adatot.

- **adatkulcs exportálása:** Ez a függvény két kulcstokent kap inputként, $\{DataKey, K_i\}_{MK}$-t és $\{KEKKey, KEK_j\}_{MK}$-t. Dekódolja mind a két kulcstokent MK-val, ellenőrzi, hogy a kulcsok típusa a várt $DataKey$ és $KEKKey$ típus-e, s ha igen, K_i-t kódolja KEK_j-vel, majd visszaadja a $\{DataKey, K_i\}_{KEK_j}$ kulcstokent. Ez kerül majd átküldésre egy másik modulnak, amely importálhatja a K_i kulcsot.

- **adatkulcs importálása:** Ez a függvény két kulcstokent kap inputként, $\{DataKey, K_i\}_{KEK_j}$-t és $\{KEKKey, KEK_j\}_{MK}$-t. Dekódolja a $\{KEKKey, KEK_j\}_{MK}$ kulcstokent MK-val és ellenőrzi a kulcs típusát. Majd a $\{DataKey, K_i\}_{KEK_j}$-t az eredményként kapott KEK_j-vel dekódolja, és ellenőrzi a kapott kulcs típusát. Végül az eredményként kapott K_i-t kódolja a mesterkulccsal. Ezután visszaadja az így kapott $\{DataKey, K_i\}_{MK}$-t.

A fent definiált egyszerű API-t a következőképpen modellezhetjük a spi-kalkulus segítségével. Jelölje $MODULE^{ENC}$, $MODULE^{DEC}$, $MODULE^{EXP}$, $MODULE^{IMP}$ rendre az adat-kódoló, adat-dekódoló, adatkulcs-export és adatkulcs-import processzeket. Minden processz kommunikációs csatornákon keresztül kapja az adatokat, ebben az esetben az argumentumokat. A fogadási kommunikációs csatornákat rendre a $c_{enc}, c_{dec}, c_{\exp}, c_{imp}$ nevek jelölik, ezeken keresztül kapják a proccesszek az adatot. Továbbá definiálunk egy $\overline{c_{user}}$ csatornát, amelyen keresztül a processzek a környezetnek (hosztnak) küldhetnek adatokat. A processzek formális leírása a következő:

1. $MODULE^{ENC}(MK)$ @
$c_{enc}(x_{data}, x_{token0}).case\ x_{token0}\ of\ \{x_{typeK}, x_{K_i}\}_{MK}\ in\ [x_{typeK}\ is\ DataKey]$
$\overline{c_{user}} < \{TData, x_{Data}\}_{x_{K_i}} >$

2. $MODULE^{DEC}(MK)$ @
$c_{dec}(x_{token1}, x_{token2}).case\ x_{token2}\ of\ \{x_{typeK}, x_{K_i}\}_{MK}, x_{token1}$
$of\ \{x_{typeData}, x_{Data}\}_{x_{K_i}}\ in\ [x_{typeK}\ is\ DataKey]\ [x_{typeData}\ is\ TData]$
$\overline{c_{user}} < x_{Data} >$

3. $MODULE^{EXP}(MK)$ @
$c_{\exp}(x_{token3}, x_{token4}).case\ x_{token3}\ of\ \{x_{typeK}, x_{K_i}\}_{MK}, x_{token4}$
$of\ \{x_{typeKEK}, x_{KEK}\}_{MK}\ in\ [x_{typeK}\ is\ DataKey]\ [x_{typeKEK}\ is\ KEKKey]$
$\overline{c_{user}} < \{DataKey, x_{K_i}\}_{x_{KEK}} >$

4. $MODULE^{IMP}(MK)$ @

$c_{imp}(x_{token5}, x_{token6}).case\ x_{token6}\ of\ \left\{x_{typeKEK}, x_{KEK}\right\}_{MK}\ ,\ x_{token5}$

$of\ \left\{x_{typeK}, x_{K_i}\right\}_{x_{KEK}}\ \ in\ \left[x_{typeK}\ is\ DataKey\right]\ \left[x_{typeKEK}\ is\ KEKKey\right]$

$\overline{c_{user}} < \left\{DataKey, x_{K_i}\right\}_{MK} >$

A teljes API-t a fenti processzek replikációinak párhuzamos kompozíciójaként reprezentáljuk, néhány kezdeti kulcstoken kiadásával. Ezek a kulcstokenek azért kerülnek kiadásra, mert ezek a HSM-en kívül tárolódnak, s ezért bárki (beleértve a támadót) hozzájuk férhet.

$Sys_{API}(K_i, KEK_j)$ @

$$(vMK)\left(\begin{array}{l}\overline{c_{user}} < \left\{DataKey, K_i\right\}_{MK}, \left\{KEKKey, KEK_j\right\}_{MK} > . \\ \left(!MODULE^{ENC}(MK) \mid !MODULE^{DEC}(MK) \mid !MODULE^{EXP}(MK) \mid !MODULE^{IMP}(MK)\right)\end{array}\right)$$

Formálisan is bizonyítható (amit helyhiány miatt most mellőzünk), hogy ez az API semilyen körülmények között nem fogja kiadni a környezete számára a kulcsokat. Az intuitív magyarázat az, hogy az egyetlen függvény, amely nyíltszöveget ad vissza az az *adat dekódolás* függvény, ám a típusindikátorok miatt az *adat dekódolás* függvény csak akkor adja vissza a nyíltszöveget ha annak tipusa *TData* , vagyis nem kulcs.

A formális bizonyítás során a következők tesztelési ekvivalenciák fennállását kell bizonyítani: $Sys_{API}(K_i, KEK_j) \approx Sys_{API}(K_i', KEK_j)$ és $Sys_{API}(K_i, KEK_j) \approx Sys_{API}(K_i, KEK_j')$ minden K_i, K_i', KEK_j, KEK_j' esetén. Ennek bizonyítása indukció segítségével történik. Felteszük, hogy kezdetben az *R* támadó processz nem rendelkezik semmilyen kulccsal, azaz a rendszer biztonságos állapotban van. Majd bebizonyítjuk, hogy ha a rendszer biztonságos állapotban van, akkor az *R* és a rendszer közötti bármely üzenetváltást követően is biztonságos állapotban marad. Ez tehát azt jelenti, hogy a támadó nem tud egyetlen kulcsot sem megszerezni a rendszerből az API-n keresztül.

Konklúzió

Az API szintű támadások komoly veszélyt jelentenek a hardver biztonsági modulokra nézve. Ebben a cikkben egy formális módszert javasoltunk az API biztonsági analízisére, mely lehetővé teszi annak bizonyítását, hogy egy külső támadó nem képes titkos kulcsot kinyerni a modulból az API-n keresztül. A sikeres bizonyítás az API biztonságosságát jelenti, míg a sikertelen bizonyítás általában valamilyen API hibára hívja fel a figyelmet. A javasolt módszer alapját a spi-kalkulus képezi, melyet eredetileg kulcscsere protokollok analízisére terveztek. Egy egyszerű API-n keresztül bemutattuk a spi-kalkulus alkalmazhatóságát. Eddigi tapasztalataink azt mutatják, hogy a spi-kalkulus jól alkalmazható API ellenőrzési célokra.

Referenciák

[1] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the Spi calculus. Technical Report SRC RR 149, Digital Equipment Corporation, Systems Research Center, January 1998.

[2] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors – a survey . Technical Report UCAM-CL-TR-641, University of Cambridge, Computer Laboratory, August 2005.

[3] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the CHES 2001 Workshop*. Springer LNCS 2162, 2001.

[4] M. Bond. *Understanding security APIs*. PhD thesis, University of Cambridge, 2004.

[5] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, October 2001.

[6] M. Bond and J. Clulow. Encrypted? Randomised? Compromised? (when cryptographically secured data is not secure). In *Proceedings of the Workshop on Cryptographic Algortihms and their Uses*, 2004.

[7] M. Bond and P. Zielinski. Decimalisation table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, January 2003.

[8] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19, July 2001.

[9] J. Clulow. The design and analysis of cryptographic APIs. MSc thesis, University of Natal, South Africa, 2003.

[10] J. Clulow. On the security of PKCS#11. In *Proceedings of the CHES 2003 Workshop*. Springer LNCS 2779, 2003.

[11] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of API-level vulnerabilities. In *Proceedings of the ACM/IEEE Conference on Software Engineering (ICSE)*, 2005.

[12] A. H. Lin. Automated analysis of security APIs. MSc Thesis, Massachusetts Institute of Technology, May 2005.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, September 1992.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC)*, June 2001.

[15] P. Youn. The analysis of cryptographic APIs using the theorem prover otter. MSc Thesis, Massachusetts Institute of Technology, May 2004.