Bezpečnost v IT

4. přednáška

Radek Janoštík

Univerzita Palackého v Olomouci

5. 4. 2024

Outline

- Aktuální (kyber)bezpečnostní situace
- Zabezpečení webových aplikací (obecně)
- Škodlivý software
- Chyby SW a HW
- Doporučená literatura

Aktuální (kyber)bezpečnostní situace

- Francouzský úřad práce napaden – odcizeny osobní údaje za 20 let (43mil osob)
  - Jména, data a místa narození, adresy, emaily, telefony, čísla soc. pojištění, ...
- Útok postranním kanálem na procesory Apple M1 a M2
  - Útok `https://gofetch.fail/` umožňuje získat privátní klíče (2048RSA kolem hodiny)
- Federální soud nařídil Google, aby identifikoval uživatele, kteří sledovali video
  - Mezi 1. a 8. lednem (asi 30tisíci)
  - Chtějí jména, adresy, telefony, případně IP adresu u nepřihlášených
  - Neví se, zda Google vyhověl
  - Důvod: Hledají uživatele `elonmuskwhm`, podezřelého z praní špinavých peněz, jemuž policista poslal odkaz na video.
- Odhalen *backdoor* v knihovně `xz-utils`
  - Příprava několik let, obfuskace, skrývání skutečné funkcionality
  - Dostal se do repozitářů velkých distribucí (Fedora, Debian testing, ...)
  - `[ebuild     UD] app-arch/xz-utils-5.4.2 [5.6.1]`

Webové aplikace – bezpečnost – motivace

- = Potenciálně zranitelné aplikace, které jsou volně dostupné
- Lákadla:
  - obsahují zajímavá data
  - poskytují zajímavé služby
  - firmy na nich mohou být závislé
- Útoky mohou být jednoduché, mnoho způsobů
- Nastavit (nějak) server a zprovoznit webovou aplikaci je snadné
- Mnoho jich však není udržováno, nereagují na „trendy“
- Za krátkým kódem se skrývá „velká mašinérie“
  - Neznalý programátor může způsobit spoustu problémů

HTTP-only webová aplikace

- Mějme nějakou (obyčejnou) webovou aplikaci, která ani nemusí mít přihlášení
- Proč bychom měli nasadit SSL?
- Kdokoliv na cestě může udělat MITM útok
- Např. vkládání reklamního obsahu do webů
- Sledování statistik přístupů
- U aplikací „s přihlášením“ je SSL téměř povinné
  - Heslo by mohl kdokoliv odchytit a přihlásit se za uživatele
  - Problematika šifrování hesla pomocí JavaScriptu
  - Viditelnost session-id, cookies

HTTP Basic Access Auth

- Protokol HTTP má základní podporu autentizace jménem a heslem
- Komunikace pomocí HTTP hlaviček
- Podpora v prohlížečích (uživatel zadá jméno a heslo)
- Server při potřebě autentizace odešle hlavičku `WWW-Authenticate: Basic realm="Restricted Area"`
- Klient musí do hlavičky každého požadavku přidávat jméno a heslo
  - Odesílá hlavičku: `Authorization: Basic jmeno:heslo`
  - `jmeno:heslo` je zakódováno v Base64
- Aplikace si musí zpracovávat hlavičky a rozhodnout, zda zobrazí obsah či ne
- Nevýhody: Heslo putuje v otevřené podobě, nemůže obsahovat dvojtečku
- Výhoda: Podpora webserverů (znepřístupnění části statické aplikace)

Nasazení SSL

- Získání „klasického“ SSL certifikátu je sto až tiscíkorunová položka (v závislosti na CA)
- Situaci změnila CA Let’s Encrypt
  - Bezplatně vydává (již) důvěryhodné certifikáty
  - Velký důraz na automatizaci nasazení (certbot, ACME.sh)
  - Apache má vestavěný modul mod_md
  - `https://letsencrypt.org/how-it-works/`
- Privátní klíč bývá šifrován $\Rightarrow$ po každém restartu služby vyžadováno heslo
  - Možné odšifrování – potom čitelný a disku
  - Pozor na nastavení oprávnění (pouze pro root: `chmod 400 server.key`)

Přihlášení uživatele

- Situace: Máme webovou aplikaci s SSL a vyplňujeme přihlašovací formulář
- Brání nám (z hlediska bezpečnosti) něco v odeslání hesla provozovateli?
- Ne, jen pouze důvěra provozovateli, že „rozumně“ pracuje s hesly
- Aplikace by měla pracovat s hesly v otevřené podobě „co nejméně“
  - Zpracování při registraci
  - Ověření při přihlášení
- V databázi by nikdy neměla být hesla v otevřené podobě
  - Unikne-li databáze z aplikace $\Rightarrow$ každý zná jejich hesla
  - Správci serveru/databáze mají přístup k heslům
- Obecně bychom (opravdu) neměli používat jedno heslo k různým aplikacím

„Rozumné nakládání s hesly“

- Varianta 1: V DB uložíme `jméno` uživatele a `hash` hesla
  - Při přihlášení spočítáme hash s hesla a porovnáme
  - Určitě lepší než plaintext
  - Jakou zvolit hashovací funkci? SHA-256? Už není považována za bezpečnou
  - Byla navržena pouze pro kontrolní součty a pro *digest*
  - Lepší funkce navržené k tomuto účelu: *bcrypt*, *scrypt*, *PBKDF2*
- Varianta 1 má problém: z uniklé DB půjde ihned poznat, že dva uživatelé mají stejné heslo
- U slabších hashovacích funkcí poměrně snadné prolomit (*rainbow tables*)
- Vygenerujeme náhodný řetězec – *sůl* pro každého uživatele
- V DB uložíme `jméno`, `sůl` a `hash(heslo+sůl)`
  - Na první pohled nelze poznat stejná hesla, prolamování je těžší
  - Např.: *PBKDF2* má sůl jako vstup

Slabá uživatelská hesla

- I skvěle zabezpečená webová aplikace může být zranitelná skrze uživatele
- Vynucování pravidel pro tvar hesla (délka, čísla, velká písmena, speciální znaky)
- Může být dvojsečná zbraň
  - Uživatelé mají tendenci mít sice silné heslo, ale na více službách
  - Dát pravidla předem na registrační formulář
- Dobrá může být kontrola výskytu ve známých slovnících
  - `https://github.com/danielmiessler/SecLists/tree/master/Passwords`
  - `https://wiki.skullsecurity.org/index.php/Passwords`
- Idea: Co tahle zpětně kontrolovat hesla v DB, zda nejsou ve slovnících?
  - Je to dobrý nápad? Proč ano? Proč ne?
- Omezování počtu přihlášení uživatele v čase (banování IP?)

Login session – session hijacking

- Po přihlášení server vygeneruje *Session Id*, které má nějakou „trvanlivost“
- Uživatelův prohlížeč posílá *Session Id* s každým požadavkem
- Server tak uživatele pozná a nemusí probíhat ověřování jménem a heslem
- Problém: Při nešifrovaném spojení můžeme odposlechnout
  - Stačí pak poslat hlavičku se *Session Id* a budeme se vydávat za uživatele
- Při krátkém *Session Id* možný bruteforce (moc se nekontroluje frekvence)
- Některé aplikace posílají *Session Id* jako parametr v URL
  - Odposlech „pohledem přes rameno“ (vyfocení)

Cross Site Request Forgery

- Uložená *Session Id* si prohlížeče pamatují do zavření/vypršení
- Uživatel se přihlásil např. na `nejaky-email.cz`
  - Získal *Session Id*, prohlížeč si jej zapamatoval
  - Při další návštěvě je přihlášen a může odesílat maily bez hesla
- Uživatel ve stejném prohlížeči otevře stránku `hack-me.cz`
- Obsahuje podvodný formulář typu:

```
<h1>Ziskejte Iphone Zdarma</h1>
<form action="https://nejaky-email.cz/sendmail.php" method="post">
    <input type="hidden" name="subject" value="Predmet emailu" />
    <input type="hidden" name="to" value="nejaka@adresa.cz" />.
    <input type="hidden" name="text" value="text emailu" />
    <input type="submit" value="Kliknete pro ziskani vyhry" />
</form>
```

Cross Site Request Forgery

- Uživatelův prohlížeč přidá zapamatovanou *Session Id* k požadavku
- Může dojít k provedení nechtěné operaci na úplně jiném serveru
- Možné odeslat i pomocí Javascriptu, případně rovnou požadavek bez formuláře
  - Přímé JS požadavky na jinou stránku prohlížeče (již) blokují
- Obrana: *Antiforgery Token*
  - Každý formulář v aplikaci obsahuje jednorázový *token*
  - Uložen ve skrytém `input`
  - Omezená platnost (časová, použití)
  - Dnešní webové frameworky mají zabudovanou podporu (např. ASP.NET)
- Obrana: Cookies s příznakem `SameSite: Strict`
  - Takto nastavený cookie by se neměl odeslat z jiné adresy
  - Nemusí podporovat všechny prohlížeče
  - Více na: Návrh RFC, `https://web.dev/samesite-cookies-explained/`

Zabezpečení souborů

- Webový server Apache ve výchozím nastavení umožňuje zobrazit obsah adresáře
- Např.: `https://apollo.inf.upol.cz/` neobsahuje spustitelný skript
  - Po zadání adresy vytvoří „webovou stránku" se seznamem souborů v adresáři
- Toto nastavení často zůstává zapnuto
- Možné zkoušet některé zajímavé adresáře: `/uploads`, `/images`, `/photos`, ...
- Získávání souborů pomocí cesty v URL
  - Mějme webovou aplikaci, která poskytuje soubory na adrese `app.cz/getFile.php?=./photos/photo1.php`
  - Aplikace často kontrolují pouze přítomnost soubory, nikoliv oprávnění
  - Pomocí úpravy URL můžeme získat např. hesla do DB, či zdrojové kódu
  - `app.cz/getFile.php?=../../config/config.php`
  - `app.cz/getFile.php?=../../../../etc/passwd`

Chybová hlášení

- Výpisy chybových hlášení do stránky je velmi dobré pro vývoj
  - Můžeme vidět kódy chyb
  - Stack Trace
- Při produkčním nasazení bychom neměli zobrazovat technické informace vůbec
- Pouze obecná chybová hlášení „pro lidi“
- Technická hlášení a stack trace logovat na souborový systém
- Čím méně toho útočník ví, tím hůře se mu bude útočit
  - Nebude znát strukturu aplikace (kde jsou které skripty)
  - Nemusí znát ani programovací jazyk (i přípona `.php`)
  - Neodhalovat strukturu databáze
  - Není potřeba odhalovat verzi webserveru a jazyk
- Některé frameworky mají přepínač v konfiguraci
  - Kompilace v release režimu (ASP.NET)

SQL injection

- Populární útok založený na špatném ošetřením uživatelských vstupů
- Tyto vstupy poté přímo vkládány do SQL dotazů

```
$query = "SELECT id, roles FROM users WHERE user='".$_POST["user"]."'
                               AND hash='".hash($_POST["pass"])."'";
if (isNotEmpty(getRowFromDb($query)) { processLogin(...) }
```

- Útočník nejdříve zadá nějaký špatný znak (apostrof, středník)
- Špatně nastavený server vypíše chybovou hlášku s celým SQL dotazem
- Útočník upraví dotaz aby vrátil výsledek i bez znalosti hesla

SQL injection

- Například nastaví jméno na: `admin' --`
  - Dvě pomlčky uvozují komentář – část za nimi se bude ignorovat
- Vzniklý dotaz

```
SELECT id, roles FROM users WHERE username='admin' -- AND hash='';
```

- Špatně ošetřený dotaz získá role admina a přihlásí nás
- Poddotaz, který si zjistí hash z databáze
- Možné obejít filtrování – `' OR 1=1`
- Destruktivní dotazy – `'; DROP TABLE users;`
- Spuštění cizího kódu (Oracle – java kód, MSQQL – `xp_cmdshell`)

SQL injection – obrana

- Nepoužívat neošetřené uživatelské vstupy do SQL dotazů
- Používat SQL prepared statements (snad každý jazyk podporuje)

```
$stmt = $mysqli->prepare("INSERT INTO test(id, label) VALUES (?, ?)");
$stmt->bind_param("is", $id, $label);
```

- Hodnoty proměnných již nejsou interpretovány jako SQL
- Omezení datových typů `"is"` – integer a string
- Nezobrazovat chybová hlášky (skrytí struktury dotazů)
- ORM – většina frameworků řeší ve výchozím nastavení

Zneužití skrytých input elementů

- Některé informace si aplikace ukládají do skrytých `input` elementů a pak nekontrolují
- Př. formulář pro upravení příspěvku

```
<input type="hidden" name="post_id" value="25" />
<input type="text" name="post_text" value="Obsah prispevku" />
```

- Pouhou změnou obsahu `post_id` bychom mohli změnit příspěvek, na který nemáme oprávnění
- ⇒ Jakýkoliv vstup (i ze skrytých elementů) musíme kontrolovat
- Extrémní případ (ne ojedinělý)

```
<input type="hidden" name="price" value="2000" />
```

- Systém bez kontroly může objednat položku za jinou cenu

FrontEnd a JavaScript

- FrontEnd aplikace by měl co nejméně prozrazovat o BackEndu
  - Nepoužívat interní objekty, ale DTO – posílat pouze potřebná data
  - Neztotožňovat názvy sloupců v DB s možností filtrace
  - Považovat všechna vstupní data z FE jako potenciálně nebezpečná
- Zdrojové kódy javascriptové části jsou veřejné
  - Každý je může studovat
  - Upravovat a zkoušet používat
  - Neměly by obsahovat citlivé informace
  - Dotazy typu: „Chtěl bych se připojit do databáze pomocí JS"
- Minifikace JS kódu jej pouze zmenšuje `https://www.minifier.org/`
- Obfuskace znesnadňuje jeho pochopení a úpravy `https://obfuscator.io/`

Cross Site Scripting (XSS)

- Zneužití nevalidování vstupu a jeho interpretace jako HTML
- Situace: Mějme návštěvní knihu s formulářem
- Útočník do formuláře vyplní:

```
Toto je ale krasna stranka!
<script>alert(document.cookie);</script>
```

- Návštěvní kniha vůbec nekontroluje vstup a výstup
  - Každému dalšímu návštěvníkovi vloží skript do DOM stránky
  - Prohlížeče daný skript interpretují a vykonají
- Cookie může být skriptem odeslán na jinou stránku
- Otravné (ztráta důvěry)

Oblíbené redakční systémy

- Od píky se dělá velmi málo webů
- Používání frameworku, redakčních systémů (WordPress, Drupal, Joomla, ...)
- Tyto systémy často nasazují laici
  - Často nastaveny výchozí jména a hesla, či velmi slabá
  - Nezabezpečení serverové části
  - Často zobrazují technické chybové hlášení (snadnější útok)
- Jsou to komplexní systémy ⇒ snadno se udělá chyba
- Provozovatelé často neaktualizují („co funguje, na to nesahej")
- V provozu obrovské množství zastaralých verzí
- Automatizované útoky (detekce CMS, verze, specializované útoky, výchozí hesla, soubory v URL, ...)
- Obrana: Aktualizovat aplikaci a její doplňky

Dobrovolné cvičení: Cvičná webová stránka

- Máme k dispozici cvičný server s webovou aplikaci
  - `http://158.194.80.100/`
  - Použity velmi moderní technologie
- Aplikace má spoustu zranitelností
- Před vnějším světem je omezen přístup pomocí HTTP basic authentication
  - Pro zvídavé: Zkuste si odchytit, jak vypadají pakety s komunikací
- V Galerii lze prohlížet obrázky, ale také nahrávat nové
- V sekci Studenti lze řadit podle zadaných sloupců
- V knize návštěv můžete zanechat svůj vzkaz
- Administrace je tajná

Dobrovolné cvičení: Co by mělo jít

- Cross-site scripting
- Získat zdrojové kódy backendu
- Upravit zdrojové kódy backendu
- Získat hesla a uživatele do administrace
- Smazat databázi
- Session hijacking
- …?

Dobrovolné cvičení: Dobrovolný úkol

- Využijte alespoň dvě zranitelnosti
- Navrhněte k nim vhodný způsob obrany
- Reset aplikace – načtením scriptu: `http://158.194.80.100/reset.php?reset=true`
  - Pokud vám něco povede, zapište si postup (pro replikaci) a resetujte aplikaci
  - Pokud ji rozbijete a skript ji nedokáže resetovat, napište mi.
- HTTP basic auth user: `test`, heslo: `Heslo`

Fungování software

- Dnešní software jsou velmi rozsáhlé balíky se stovkami binárních souborů
- Jak můžeme zaručit, že software dělá jen to, co uživatel chce?
- Můžeme například ověřit původ software (digitálně podepsané binární soubory, dll)
- I přes to není zaručeno, že SW bude dělat co má
  - Může nadměrně vytěžovat zdroje
  - Být nestabilní
  - Obtěžovat uživatele dialogovými okny, . . .
- Důležitá je příčina nezamýšleného chování
  - Chyba v SW, špatná optimalizace
  - Špatný návrh SW (UI, „flow“)
  - (Zlomyslný) záměr vývojářů
  - Napadení SW, podvržení

Malware

- Dnes se budeme zabývat pouze úmyslnou (nechtěnou) funkcionalitou
- Malicous Software ⇒ Malware
  - = SW, který dělá něco, co uživatel nechce (by nechtěl), aby dělal
- Co vše může škodlivý software dělat?
- Vše, na co má oprávnění uživatel, který jej spustil
  - Síťová komunikace
  - Zatěžování zdrojů (dnes populární těžba kryptoměn)
  - Zobrazování reklamy, vyskakovací okna
  - Obtěžování uživatele (zabíjení programů, „psaní na klávesnici“, . . .)
  - Čtení/mazání disků
  - Odposlech (snímky obrazovky, stisky kláves, běžící programy, . . .)
- Spousta lidí si neuvědomuje, jakou důvěru v SW vkládají

Typy škodlivého software (1 / 2)

- Vir – program, který modifikuje ostatní programy
  - Vkládá do nich škodlivý kód
  - Často i sebe sama ⇒ šíření
  - Existence po dobu nakaženého (*transient*) nebo samostatně (*resident*)
- Trojský kůň – za chtěnou funkcionalitou schovává ještě nějakou nechtěnou
  - Často malé utilitky (editace videa, stahování, . . .)
  - Ale jiný záměr – šíření jiných virů, odposlech, . . .
- Zadní vrátka(backdoor) – skrytá funkcionalita programu
  - Obejití bezpečnostních mechanismů, nečekaný přístup
  - Většinou implementovány vývojáři nebo potají přidány do zdrojových kódu
- Červ – vir, který šíří své kopie přes síť
- Spyware – špehovací SW (klávesnice, zvuk, kamera!, citlivé fotky)

Typy škodlivého software (2 / 2)

- Adware – zobrazování reklamy uživateli
  - Často napadení prohlížeče – modifikace kliku na odkaz
  - Vkládání kódu do stránek – uživatel nepozná zdroj reklam $\Rightarrow$ zisk k útočníkovi
- Ransomware – způsobí škodu uživateli a požaduje výkupné za napravení
  - Dnes velmi populární – platba v kryptoměnách (napadení společnosti Canon 2020)
  - Otázkou je, zda ransomware „dodrží slovo“
  - Často se pouze tváří, že provedl napravitelnou operaci
  - I jiné výkupné než peníze (Nvidia 2022)
- Kontrola počítače – často tiché vyčkávání na *trigger*
  - poté spuštěn škodlivý kód – botnety
- *Jokeware* – software pro „vystřelení si z kolegy“ – nekomerční původ virů
  - Např. *fork bomba*
- Velmi často je škodlivý SW kombinací více typů – šíření + škoda (+ touha po zisku)

Stručná historie malware

- Myšlenka škodlivého SW se nezrodila v 90. letech
- Známé i výrazně dříve
  - Fred Cohen 1984 – článek COMPUTER VIRUSES: THEORY AND EXPERIMENTS
  - $\approx$ 1979 zmínky o škodlivém kódu, který vkládá kompilátor (U.S. Air Force)
- V 90. letech pouze rozšíření (boom počítačů)
  - Později boom s příchodem internetu
  - Dnes internet hlavním kanálem pro šíření
- 1949 – John Von Neumann – první program, který reprodukuje sebe sama
- Existuje nějaký teoretický základ pro „reprodukci programu“?
  - Věta o rekurzi – 1939 – Stephen Kleene
  - Quine – program, který vypíše svůj kód a skončí

Kvalitní virus (z pohledu útočníka)

- Obtížná detekce – uživatelem i antivirovými programy
  - Vytížit všechny zdroje? Nebo jen někdy?
  - Jak moc okatě dělat svůj záměr?
- Obtížné zneškodnění – stačí odstranit „pár" instrukcí z binárního souboru?
  - Morfovací techniky, „zažrání se do systému"
- Rychlé šíření (lokální i síťové)
- Náročnost vytvoření
  - Jednoduchý vir na milion počítačů („třeba to vyjde")
  - Komplexní vir cílený na specifickou platformu
- Platformová nezávislost – omezení OS, HW architektury
- (Hlavní) užitečnost pro tvůrce (monetizace)

Šíření malware

- Dříve vyměnitelná media (pirátské kopie programů)
- Přelom tisíciletí – email – přílohy (`prezentace.exe`, `lechtiveVideo.exe`, ...)
  - Lehká blokace – spam filtr, emailový klient
  - I přes to velmi populární a účinné (i dnes – PDF přílohy)
- Pirátský SW přes internet
- Sociální inženýrství – přesvědčení osob, že jste někdo jiný
  - Phishing – falešná technická podpora
  - Falešná instalační média (fyzický disk poštou)
- Šíření přes sdílené disky (Samba)
- Chyby v síťových aplikacích
  - Bezpečnostní chyby ve webových aplikacích
  - Chyby v síťových démonech

Šíření viru v systému

- Samotný virus na disku nedělá nic – musí se spustit
  - Důležité je první spuštění
  - Pak už může virus pracovat na svém šíření
- Připojení svého kódu před cílový program (obrázek)
  - Stačí rozumět hlavičkám binárních souborů
  - Naleznou první instrukci cílového programu a vložit před ni svůj kód
  - Ideální po svém kódu spustit i cílový program (uživatel nemusí poznat nákazu)
  - Útočník nemusí znát cílový program – univerzální
- Obklíčení cílového programu – škodlivý kód při spouštění a ukončení
  - Cíl: snížení detekce viru – může být v šifrovaném souboru
  - Pří spuštění pouze obecný kód pro rozšifrování
  - Načten do paměti, spuštěn, smazán z disku
  - Při ukončovaní – uložení šifrované kopie na disk
- Integrace do aplikace
  - Horší detekce, nutná znalost konkrétní aplikace

Oblíbené cíle virů

- Boot sektor – OS je také program na disku – něco jej musí spustit
  - Firmware počítačů načte fixní délku kódu z boot sectoru a spustí jej
  - Normálně by se začal načítat OS a všechny jeho komponenty
  - Nakazí-li virus boot sektor může převzít absolutní kontrolu nad strojem
    - Může přemostit jakékoliv systémové volání
    - Simulace zdravého boot sektoru (obtížná detekce)
- *Resident routines* – funkce OS, které jsou trvale načtené v paměti
  - Nenačítají se z disku znovu a znovu (šetření času)
  - Např. obsluha klávesnice, ošetření chybových stavů, . . .
- *Excel makro viry* – krátký vir, spouštějící dlouhé makro
- Knihovny – některé široce rozšířené (kauza log4j)
- Samotný operační systém (infiltrace škodlivého kódu do linuxového jádra)

Detekce malware

- Jak rozpoznat, že daná aplikace je škodlivá či nikoliv?
- Jak byste programovali antivir?
- Kód viru musí být někde uložen a po spuštění musí být v paměti
- *Nějak* jsou spouštěny, *nějak* se šíří
- *Vzor*, *signatura* specifických virů $\Rightarrow$ můžeme prohledávat disk, paměť
  - Pro čtení paměti je potřeba administrátorské oprávnění
  - Často stejné umístění v aplikacích (prvních x bajtů)
  - Kontrola změny velikosti/kontrolního součtu aplikací
  - Podezřelá první instrukce – `JUMP`
- Nutná aktuální databáze *vzorů, signatur*
- Často těžké rozlišit od chtěné aplikace

Odstranění malware

- Po detekci je nutné zastavit všechny běžící instance (zastavit šíření)
  - Kontrola v *nouzovém režimu*
  - Kontrola ještě před bootem OS
- V některých případech triviální – odstraníme několik bajtů z binárního souboru
  - Původní aplikace zůstane plně funkční
- Co když je vir „zažraný“ do více částí aplikace?
  - Podaří se nám odstranit všechny kusy?
  - Bude pak aplikace funkční?
- Nejlépe pak přijít na cestu, kudy se vir do systému dostal

Prevence

- Zodpovědné chování uživatelů na síti (správců/adminů)
  - Technicky můžeme být zabezpečení jakkoliv, ale nejslabším článkem jsou uživatelé
  - Hesla na nástěnce vedle PC, půjčování přístupových karet, „zastupitelnost“
- Používání sotfwaru z důvěryhodných zdrojů (podepsané)
  - *Reproducible builds* otevřeného software
  - vs. obrazy dockeru, flatpaku, ...
- Testování méně důvěryhodného softwaru na odděleném stroji (k tomu určeném)
  - Bez disku, sítě
- Kvalitní antivirový SW – dnes už není „hloupý“ skener souborů
  - Aktivní skenování paměti, systémových volání
  - Firewall
  - Napojení se do webového prohlížeče, mailového klienta
  - Pozor! Někdy dělají i HTTPS MITM útok
- Co když je antivir virem?

Taktiky proti detekci

- Po rozšíření prvních antivirů na ně začali tvůrci virů reagovat
- Hon na kočku a myš odstartoval
  - Vynalezeny různé taktiky maskování škodlivého kódu
  - A proti-taktiky, . . .
- Šifrování viru (na disku)
- Polymorfní viry – mění(mutují) sami sebe
  - Možná mutace své *signatury* $=$ horší detekce
  - *signatura* mutování?
  - Obfuskace kódu (vkládání zbytečných operací)
  - Změna fixních řetězců, pořadí operací

Intrusion Detection System (IDS)

- Detekční systémy podezřelého provozu na síti/stanici
- Host-based – běžící a sledující konkrétní stroj
- Network-based – monitorují celou síť (např. HW sonda)
- Kontrola používání konkrétních portů
- Kontrola vytížení stanice
  - Použití podobných mechanismů jako antiviry (signatura chování)
  - Většinou možné definovat vlastní pravidla
  - V případě aktivace pravidla informování správce
- *Intrusion Prevention Systems* – mohou i aktivně zasáhnout (zavřít porty, zabít aplikaci)
- Dopad režie kontrolního systému na propustnost

Honeypot

- Falešná aplikace/zařízení sloužící jako návnada „hrnec s medem“
- Plně simuluje prostředí a nechá se napadnout
  - Získává informace o činnosti viru
  - Napomáhá v budoucí obraně
- Český projekt *Honeypot as a Service(HaaS)*
  - `https://haas.nic.cz/`
  - Počátky při projektu Turris
  - Spuštění proxy, útočník komunikuje s jiným serverem
- Další systémy např. `Pentbox`, `OpenCanary`

Složitost SW

- Komplexnost software raketově roste
  - Od krátkých utilit pro jeden účel přesun ke komplexním systémům
- (Bezmyšlenkové) používání velkého množství knihoven
  - Důvěryhodnost knihoven
  - Udržovanost jejich kódu
- Používání protokolů bez jejich hlubší znalosti
- Naštěstí roste kvalita podpůrných nástrojů pro programátory
  - Bezpečný přístup do paměti, GC
  - Spoustu chyb umí odhalit i kompilátor
  - Automatické testování kódu (fuzzing)
- Synchronizační mechanismy pro paralelní programování

Bezpečnostní chyby

- = chyby v SW, které umožní programu/uživateli dělat, na co nemá oprávnění
- *Arbitrary code execution (ACE)* – spuštění libovolného kódu na počítači či v procesu
- *Remote code execution* – vzdálené spuštění libovolného kódu na počítači
- Eskalace práv – pomocí chyby v aplikaci navýšení práv uživatele (root, kernel)
- Neoprávněný přístup do paměti – aplikace je schopna číst/zapisovat do paměti jiných procesů
  - Velký problém – v paměti mohou být hesla, privátní klíče
- „Útěk z kontejneru“ – aplikaci běžící v kontejneru se podaří porušit barieru
- *Exploit* = kód, data, která využívají zranitelnosti v aplikaci pro svůj prospěch
  - Pro známe chyby dostupné na internetu – motivace záplatovat?

Správa chyb

- Tutlání bezpečnostních chyb není dobrá taktika
- Common Vulnerabilities and Exposures (CVE) – informování veřejnosti o chybách
  - Každá bezpečností chyba dostane číslo (spravuje MITRE + CNA – velké projekty)
  - Zveřejněny bezpečné detaily např.: CVE-2014-0160
  - Nebo NVD - NIST
  - CVSS – každá chyba číselně ohodnocena dle závažnosti
- Často vývojáři oznámí, že „je nějaká chyba“
  - Správci se mohou připravit na záplatování, informace se rozšíří
  - Vývojáři vydají opravy, až po nějaké době se vydají detaily, zveřejní exploity, testy
- BugBounty programy velkých projektů
- Jak se tedy zachovat při objevení bezpečnostní chyby?

Buffer overflow

- Mějme program běžící pod rootem s následujícím kódem:

```
void A() {
    char B[128];
    printf("Type log message:");
    gets(B);
    writeLog(B);
}
```

- Vyzve uživatele, aby napsal na standardní vstup, co se má zapsat do logu
- Na první pohled je všechno v pořádku, ale obsahuje chybu
  - Funkce `gets(B);` nekontroluje délku bufferu, kam zapisuje
  - Umožní tak uživateli přepsat paměť procesu

Buffer overflow

- Jak vypadá zásobník programu? (obrázek)
- Při zápisu do bufferu můžeme přepsat paměť předchozích funkcí
- V lepším případě aplikace *pouze* spadne
- V horším případě (inteligentní útočník) může spustit/přepsat libovolný kód
- Můžeme změnit návratovou adresu a „skočit“ úplně jinam do kódu
- Obrana
  - *Stack canaries* – před návratovou hodnotu umístěno náhodné číslo, kontrola
  - Data execution prevention – **NX bit** – rozdělení paměti na data a kód

Format string attack

- Mějme kód:

```c
int main(int argc, char ** argv) {
    printf(argv[1]);
}
```

- `printf()` má jako první argument formátovací řetězec
  - Nekontroluje se, zda je počet dalších argumentů roven počtu *specifikátorů*
  - Hodnoty bere rovnou ze zásobníku (předpokládá, že tam jsou)
- Při zadání formátovacího řetězce `"%x %x %x"` dojde ke čtení zásobníku
- Zápis je (překvapivě) také možný pomocí `"%n"`
  - Na příslušnou adresu zapíše počet již vygenerovaných znaků
- Obrana: `printf("%s", argv[1])`

Integer overflow attacks

- Celočíselná aritmetika bývá prováděna s čísly s omezenou délkou (8, 16, 32, 64 bitů)
- Při sčítání a násobení dvou velkých čísel může hodnota *přetéct* do záporných
  - Aplikace v jazyce C často netestují
- Mějme grafický program s argumenty pro výšku a šířku obrázku (příkazová řádka)
  - Zadáme velké rozměry obrázku (ale korektní)
  - Program si spočítá velikost potřebné paměti pro `malloc()`
  - Výpočet mu přeteče a ve skutečnosti naalokuje méně paměti
  - Zbytek obrázku přepíše paměť procesu, do které by zapisovat neměl
- $\Rightarrow$ Buffer overflow attack

Command injection attacks

- Často jsou programátoři líní a chtějí si usnadnit úkol zavoláním příkazu *shellu*
- Např kopírování souboru:

```
int copyFiles() {
    char src[100], dst[100], cmd[205] = "cp ";
    printf("Enter name of source file: ");
    gets(src);
    strcat(cmd, src);
    printf("enter name of destination file: ");
    gets(dst);
    strcat(cmd, dst);
    system(cmd);
}
```

- Vygenerovaný příkaz např.: `cp soubor1.txt novy.txt`
- co když útočník zadá jako cíl: `"novy.txt; rm -rf /"`

Time of check to Time of use attack (symlink attack)

- Mějme program běžící pod rootem zapisující do souboru:

```
int fd;
if (access("./file", W_OK)!=0 {
  exit(1);
}
fd = open("./file", O_WRONLY);
write(fd, userInput, sizeof(userInput));
```

- Uživatel může mít právo pro zápis do souboru `file`
- Kontrola oprávnění a zápis jsou časově odděleny
- Mezitím může uživatel změnit `file` za symbolický odkaz, který odkazuje někam jinam (třeba do systémových adresářů)
- Obrana – otevřít soubor rovnou a poté zkontrolovat oprávnění funkcí `fstat()`

XML útoky

- Zpracování XML souborů není triviální záležitost
- Standard obsahuje několik záludností – mohou být použity pro útoky (špatný parser)
- Využití `XML entity` pro útoky
- Např. čtení něčeho, co nikdy neskončí:

```
<?xml  version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM  "file:///dev/random" >]>
<foo>&xxe;</foo>
```

- Případně čtení `file:///etc/passwd`

Složitost HW

- Procesory obsahují čím dál více transistorů a instrukcí
  - Intel 8080 – 4500
  - Intel 286 – 134 000
  - Intel Pentium – 3 100 000
  - AMD Threadripper 2990WX – 19 200 000 000
- Instrukční sada již není hardwarová, ale emulovaná
  - Abstrakční bariéra mezi instrukcemi a HW vykonáváním
  - *Mikrokód* – interní instrukční sada s *mikroinstrukcemi*
- Možné mikrokód aktualizovat a tím zvýšit výkon (zpětně)
  - Případně „zalepit“ bezpečnostní chybu
- Dnes již možné mít „operační systém“ uvnitř procesoru, přístup k veškeré paměti, datům
  - Minix uvnitř Intelu

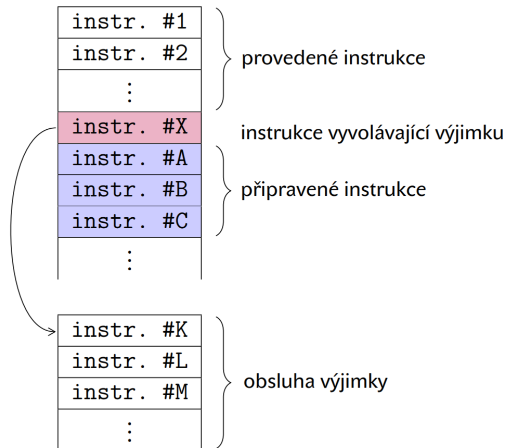
Chyby v HW

- Stejně jako s komplexností SW roste pravděpodobnost chyby
  - Tak i s komplexností HW roste pravděpodobnost chyby
- Problém může být s jejich opravením
- Např. Floating point divison bug procesorů Intel Pentium
  - Chyba při dělení – dává jiné výsledky
  - Vznikla při optimalizaci a špatném nastavení algoritmu při výrobě
  - Intel „svolával chipy k opravě“ – náklady 475 milonů dolarů
- Některé chyby lze opravit aktualizací mikrokódu, některé jsou chyby designu
- Bezpečnostní chyba v návrhu CPU = nejzávažnější chyby vůbec

Meltdown

- Závažná chyba procesorů Intel (1995 a novější)
- Pro větší rychlost nejsou mikroinstrukce prováděny sekvenčně ale paralelně
  - Mimo pořadí – *out-of-order*
- CPU se postará o synchronizaci
- CPU předpokládá, že instrukce skončí *dobře*
  - Připravuje si data pro další instrukce (rychlost)
  - U připravených dat jsou ignorovány kontroly oprávnění pro přístup do paměti
- Po dokončení obsluhy výjimky můžeme postranní kanálem (čas) zjistit, na jakou adresu se skutečně přistupovalo
  - A poté si data přečíst

Metltdown




```
; rcx = kernel address
; rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

Meltdown v pseudo-C

```
unsigned char *ptr = /* 0x12..78 */; // adresa hledanych dat
unsigned char value = 0; // prectena hodnota
unsigned char probe_array[256 * 4096]; // pomocne (velke) pole

int main() {
   signal(SIGSEGV, handle_sigsegv);// inicializace
   clflush();
   // precte byte z pameti,
   // muze skoncit vyvolanim vyjimky
   unsigned char t = *ptr;
   // pristup do probe_array, bude proveden mimo poradi
   unsigned char x = probe_array[t * 4096];
   value = t; // pokud nedoslo k vyjimce
:read_complete
   // na toto misto se vrati kod z obsluhy vyjimky
   printf("Hodnota: %i\n", value);
   return 0;
}
```

Meltdown v pseudo-C

```
// funkce osetrujici vyjimku vzniklou pri neplatnem pristupu do pameti
void handle_sigsegv(int signum) {
   for (int i = 0; i < 256; i++)
     if (is_cached(probe_array + i * 4096)) {
        value = i;
        break;
   }
// vrati se do puvodni casti
 goto read_complete;
}
```

- Takto můžeme číst paměť jádra ⇒ všech procesů rychlostí 100 - 500 kB/s
- Virtuální stroj může číst data jiného virtuálního stroje

Meltdown

- Posledním dílem skládanky je funkce `is_cached`
- Na základě rychlosti přístupu do paměti zjistíme, která stránka se reálně načetla do cache a která ne

```
int is_cached(unsigned char *ptr) {
    unsigned long long ts1 = rdtsc();
    unsigned char x = *ptr;
    unsigned long long ts2 = rdtsc();
    return (ts2 - st1) < THRESHOLD;
}
```

Doporučená četba

- McClure S., Scambray J., Kurtz G.: Hacking Exposed 7: Network Security Secrets and Solutions (7th. edition). CompuMcGraw Hill, 2012. ISBN 978-0071780285
  - Kapitola 10 – Web and Database hacking
- https://owasp.org/www-community/attacks/DOM_Based_XSS
  - XSS
- Tanenbaum A., Bos H. – Modern Operating Systems Global Edition, Pearson. 2015. ISBN 1-292-06142-1, 978-1-292-06142-9
  - Kapitola 9 – Security (593–711)
- https://meltdownattack.com/
  - Meltdown paper
  - Spectre paper
- Krajča P. Magazín katedry informatiky – 8. číslo – Meltdown
  - https://soubor.inf.upol.cz/public/Magazin-KI/magazin-ki-08.pdf