

B4B350SY: Operační systémy

Lekce 8: Bezpečnost (security)

Michal Sojka

michal.sojka@cvut.cz



November 12, 2025

1 Základní koncepty kybernetické bezpečnosti

- Co je to bezpečnost?
- Cíle zabezpečení
- Zabezpečení OS
- Mechanismy a politiky
- Zásady bezpečnostního designu

2 Řízení přístupu

3 Stack overflow

- Co je přetečení zásobníku?
- Útoky a ochrany proti nim

Cíl přednášky

- Vysvětlit vám základní koncepty používané v kybernetické bezpečnosti
- Předat znalosti potřebné pro vyřešení (nepovinné) úlohy – útok pomocí přetečení zásobníku

Co je to bezpečnost?

■ Pro každého něco jiného



Istalk



Počítačová (kybernetická) bezpečnost

Information security

- Ochrana **mých zájmů** (počítačem ovlivnitelných) před nepřátelskými hrozbami
- Velmi individuální a subjektivní
 - Různí lidé mají různé zájmy
 - Různí lidé čelí různým hrozbám
- Neexistují univerzální řešení
 - Je počítač s Windows dostačující k uložení přísně tajných informací?
 - Je připojen k internetu? Kdo k němu má přístup – fyzicky i vzdáleně?
 - Tvrzení, že systém je bezpečný má smysl jen vzhledem k dobře definovaným **cílům zabezpečení**, které definují
 - **hrozby** (tj. proti čemu jsme systém zabezpečili – např. napadení známým virem) a
 - **bezpečné stavy** systému (tj. za jakých podmínek je systém považován za bezpečný – např. antivirus a firewall jsou zapnuty)

Cíle zabezpečení

- Definují **co** má být chráněno, **proti komu** a za jakých **podmínek**.
- Před nasazením každého systému se nad tím aspoň zamyslete!
 - Často je potřeba to mít sepsané a schválené (od zákazníka, šéfa, ...).
- **Co** má být chráněno se často vyjadřuje pomocí tzv **CIA** vlastností:
 - **Utajení, důvěrnost** (**C**onfidentiality)
 - X se nesmí dozvědět o Y
 - Příklad: Franta se nesmí dozvědět administrátorské heslo.
 - **Integrita**
 - X nesmí narušit Y
 - Příklad: Karel nesmí měnit libovolné záznamy v databázi uživatelů.
 - **Dostupnost** (**A**vailability)
 - X nesmí způsobit nedostupnost Y pro Z
 - Příklad: Žádný uživatel nesmí způsobit pád web serveru.

Typické prostředky pro zajištění CIA vlastností

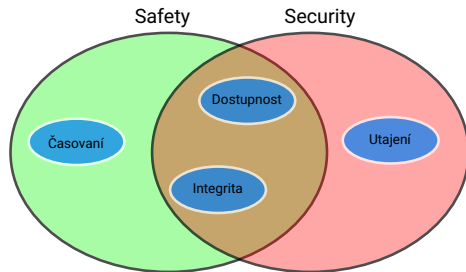
		What to protect?		
		Confidentiality	Integrity	Availability
Perspectives	Cryptography	Encryption	Digital Signatures Hashing/MAC	
	Network Security		TLS VPN Routing, Firewalling	Replication
	Operational Security		Password Strength Training, Best Practices, Certified Processes Reproducible Builds	Audit
	Posthumous Security		Malware Analysis Honeypots Log Anomaly Analysis	Forensics, Debugging
	Reactive Security	Taint Tracking	Antivirus Lockstepping	Watchdog
	Attack Mitigation		Intrusion Detection ASLR Stack-Smash Protection Pointer Obfuscation	Log Monitoring
	Resilience		Formal Verification Compartmentalization, Sandboxing, Virtualization Secure Boot	Redundancy Self-Healing

Source: <https://genodians.org/nfeske/2019-07-11-security>

Více druhů bezpečnosti

- Angličtina rozlišuje dva termíny „safety“ a „security“, které se do češtiny (i jiných jazyků) překládají oba jako bezpečnost.
- **Safety** – ochrana okolí před systémem
 - letadlo nezpůsobí škody v okolí (smrt lidí, škody na majetku)
- **Security** – ochrana systému před okolím
 - hacker neovládne vaše auto
 - teroristé nezpůsobí pád letadla, srážku vlaků

Vztah cílů zabezpečení k safety a security (výjimky existují):



Současný stav zabezpečení OS

■ Historicky:

- Zaostával za vývojem potřeb uživatelů
 - Např.: Bezpečnostní řešení byla zaměřena na firemní („enterprise“) zákazníky
- Zaostával za objevujícími se hrozbami
 - Zaměření na ochranu uživatelů mezi sebou (práva k souborům na disku), ne na ochranu uživatelů před nedůvěryhodnými aplikacemi

■ V některých ohledech se zlepšuje:

- Např. OS chytrých telefonů používají důkladnější zabezpečení než desktopy
- Objevuje se méně kritických bezpečnostních děr v běžných OS

■ V jiných se zhoršuje:

- Velikost, funkcionalita a složitost OS stále roste
- Jen málo lidí skutečně ví, jak psát bezpečný kód
- Stále více lidí umí na systémy útočit

Bezpečnost operačních systémů

- Co by mělo být cílem OS v oblasti bezpečnosti?
- Minimálně:
 - poskytovat **mechanismy** umožňující tvorbu bezpečných systémů,
 - které jsou schopny bezpečně implementovat uživatelem či administrátorem nastavenou **politiku**
 - a to tak, aby tyto mechanismy nebylo možné obejít.
- Bezpečnost systému je tak silná, jak silný je **nejslabší článek**.
 - Ďábel je skryt v detailech
 - ...i proto vás učíme assembler :-)

Dobré mechanismy zabezpečení

- Jsou široce použitelné
- Je snadné je správně a bezpečně použít
- Dají se snadno správně implementovat i verifikovat
- Nejsou v rozporu s jinými (nebezpečnostními) prioritami – např. s produktivitou práce.
- Podporují obecné principy bezpečnosti (viz slide 20)

Běžné mechanismy zabezpečení v OS

- **Systémy pro kontrolu přístupu** – kontrola, k čemu může daný proces přistupovat (např. práva k souborům)
- **Autentizační systémy** – potvrzení identity toho, jehož „jménem“ proces běží (login name, heslo, ...)
- **Logování** – kvůli auditům, detekci útoků, vyšetřování a obnovu
- **Šifrování souborových systémů** – HW lze šifrovat celý disk, SW jen souborový systém (nelze šifrovat partition table)
- **Správa pověření** (credentials)
- **Automatické aktualizace**

Bezpečnost je „prorostlá“ celým systémem. Neexistuje jen jedna komponenta zodpovědná za bezpečnost.

Rozdíl mezi politikou a mechanismem

- Politiky doprovázejí mechanismy
 - Politika **řízení přístupu** (access control)
 - Určuje, k čemu mají přístup sekretářky a k čemu vývojáři.
 - Politika **autentizace**
 - Určuje, jestli je heslo o 5 znacích dostatečné pro přístup do KOSu.
- Politika často omezuje použitelné mechanismy
 - Nestačí heslo, je potřeba se prokázat certifikátem
- Co někdo považuje za politiku, ostatní za mechanismus :-)

Zásady bezpečnostního designu

Saltzer & Schroeder [SOSP '73, CACM '74] [link]

- **Úspornost (jednoduchost) mechanismů** – keep it stupid simple (KISS)
- **Bezpečné výchozí nastavení** – nikomu se nechce trávit čas nastavováním (vždy dobrá inženýrská praxe)
- **Kompletní zprostředkování** – kontrola, že každý přístup k datům/prostředkům je autorizovaný.
- **Otevřený návrh** – ne „security by obscurity“
- **Oddělení pravomocí** – možnost dát uživatelům jen ta práva, která potřebují (ne, jen běžný uživatel, který nemůže „nic“ a root/admin, který může vše)
- **Nejmenší pravomoci** – uživatelé nemají žádná oprávnění, která nepotřebují
- **Co nejméně společných mechanismů** – minimalizace sdílení dat, minimalizace závislostí (knihoven, balíků)
- **Psychologická přijatelnost** – pokud se to těžce používá, nikdo to používat nebude
 - např. ve starších Windows měl většinou každý uživatel přiřazenou administrátorská práva, protože nebyl jednoduchý způsob, jak tato práva získat jen pro jednu operaci. Škodlivý program je tím pádem mohl automaticky zneužít.
- **“Nevěřit” slepě vstupům od uživatele** – důsledná kontrola dat předávaných z „venku“ do vnitřních částí systému
 - SQL injection, Cross-site Scripting (XSS), ...

Důvěra & Trusted Computing Base

- Všechny systémy obsahují entity, kterým se **věří**
 - pokud selžou, systém nemusí být systém bezpečný
 - hardware, OS, administrátor serveru, ...
- **Trusted Computing Base (TCB):**
 - množina všech takových entit
 - Co je součástí TCB při práci s internetovým bankovníctvím?
- Bezpečné systémy musí mít **důvěryhodné** TCB
 - **minimalizace TCB je klíčem k důvěryhodnosti.**

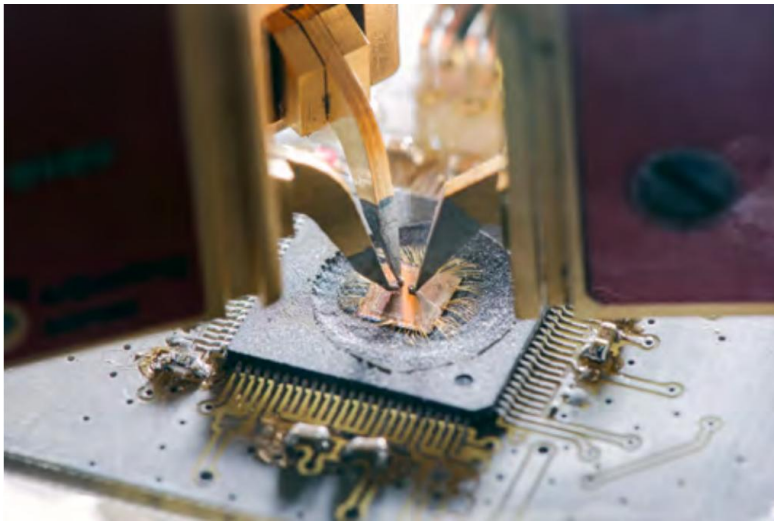
Jeep hack

Miler & Valasek, 2015



- Služba v „infotainment“ jednotce byla přes D-Bus omylem dostupná z internetu
- Útočníci nahráli SSH klíč, spustili SSH server, přeprogramovali řadič sběrnice CAN, aby šel ovládat přes sériovou linku z infotainment jednotky, ...
- <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>

Útoky jsou sofistikované



Využití postranních kanálů – zde se hledají šifrovací klíče pomocí měření elektromagnetického pole v okolí čipu

Souhrn

- Bezpečnost je velmi subjektivní, jsou potřeba dobře **definované cíle zabezpečení**
- Bezpečný OS by měl poskytovat:
 - dobré bezpečnostní **mechanismy**
 - podporující různé uživatelské **politiky**.
- Bezpečnost je daná **důvěryhodností** klíčových entit
 - **TCB**: množina všech klíčových entit
 - Jádro OS je nezbytně součástí TCB

Mechanismy a politiky pro řízení přístupu

Podrobnější pohled na jeden z aspektů zabezpečení OS.

- **Politika**

- Specifikuje, kdo má povolen přístup k čemu
- a jak se to může měnit v čase

- **Mechanismus**

- Implementuje politiky (viz dále)

- **Některé mechanismy nabádají k některým politikám**

- Některé politiky nejdou vyjádřit pomocí mechanismů ve vašem OS

Základní princip

Matice řízení přístupu [Lampson'71] definuje **stav ochrany** v daném čase

- Objekty jsou např. soubory
- Subjekt je např. uživatel
- Subjekty mohou být zároveň objekty

	Obj1	Obj2	Obj3	Subj2
Subj1 (alice)	R	RW		send
Subj2 (bob)		RX		control
Subj3 (root)	RW		RWX	recv

Ukládání stavu ochrany

- Typicky ne jako matice
 - moc „řídké“, neefektivní, dynamické
- Dvě zřejmé volby:
 - 1 Ukládání jednotlivých sloupců dohromady s objektem
 - Každý sloupec je nazýván „seznam pro řízení přístupu“ (**access control list**, ACL) daného objektu
 - 2 Ukládání jednotlivých řádků dohromady se subjektem
 - Definuje objekty, ke kterým má daný subjekt přístup – doména ochrany (**protection domain**) daného subjektu
 - Každý takový řádek je nazýván „seznam schopností“ (**capability list**)

Seznamy pro řízení přístupu (ACL)

- Implementováno skoro ve všech běžných OS
- Subjekty jsou obvykle sloučeny do tříd
 - např. v UNIXu: majitel, skupina, ostatní

```
$ ls -ld /var/spool/cups
```

```
drwx--x--- 1 root lp 6754 Nov 22 00:00 /var/spool/cups
```
 - obecnější seznamy ve Windows či v současném Linuxu (příkazy `get/setfacl`)
 - mohou obsahovat „negativní“ oprávnění – např. pro vyloučení přístupu několika uživatelů ze skupiny
- Meta-oprávnění
 - řízení členství ve třídách
 - dovolují modifikaci ACL

Obj1

Subj1	R
Subj2	
Subj3	RW

Schopnosti (capabilities)

- **Schopnost** [Dennis & Van Horn, 1966] je prvek seznamu schopností

Subj1 (alice)

Obj1	Obj2	Obj3	Subj2
R	RW		send

- **Pojmenovává** objekt, aby s ním program mohl zacházet (Obj2)
- **Uděluje** k objektu práva (RW)
- Všichni, kdo vlastní „schopnost“ mají právo s objektem pracovat
- Použití:
 - Méně časté v běžných systémech (ale pomalu se to mění)
 - KeyKOS (VISA transaction processing) [Bomberger et al, 1992]
 - Capsicum capabilities (FreeBSD)
 - Častěji ve výzkumných OS: NOVA, EROS, L4Re kernel
 - Mnohé z těchto systému se nyní začínají prosazovat i v komerční oblasti
- Pozor: V Linuxu má pojem “capabilities” jiný význam, než co popisuje tento slide

ACL vs. schopnosti

Seznamy řízení přístupu (ACL)

- Tradiční, všeobecně známé, ale...
- Proces musí být schopen **zjistit jaké objekty existují** (pojmenování) a pak teprve je může používat (a nebo mu je k nim přístup odepřen)
- Typicky to řeší tzv. **ambientní autorita** – tj. každý proces má všechna práva uživatele, který ho spustil (např. „vidí“ celý souborový systém a může zjistit, kteří další uživatelé jsou v systému.
 - Pokud program spouští jiný program, potomkovi nelze jednoduše práva omezit.
 - V Linuxu se dnes tento problém řeší pomocí „jmenných prostorů“ (**namespaces**), ale není to elegantní a trpí to některými nedostatky

Schopnosti

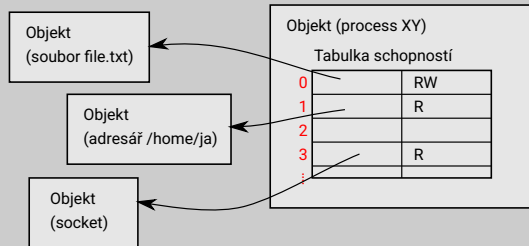
- Lepší vlastnosti z pohledu bezpečnosti.
- Neexistuje ambientní autorita, každému procesu jsou **delegovány** jen ty **schopnosti**, které potřebuje (zásada nejmenší pravomoci).
- Např. proces nevidí všechny soubory, ale jen soubory (či celé adresářové stromy), které mu rodič nebo nějaká služba „delegoval(a)“.
- Nikdo nemůže delegovat schopnosti, které sám nemá.

Možná implementace a použití schopností

Uživatelský proces XY

```
write(0, "Hello");
int program;
while ((program = readdir(1)) != -1) {
    if (getname(program) == "myprog.exe") {
        // spust' /home/ja/myprog.exe a deleguj mu objekt socket na indexu 2
        child = create_process(program, capabilities=[-1, -1, 3]);
    }
    revoke(program);
}
```

Jádro OS



- Schopnosti jsou podobné jako „file descriptor“ v UNIXových OS.
- Program identifikuje schopnosti číslem.
- Schopnosti jdou získat jen tak, že nám je někdo deleguje. Např.:
 - Rodič deleguje při vytváření potomka přístup jen k jednomu pracovnímu adresáři a stdout.
 - Služba souborového systému deleguje procesu přístup k souboru v rámci odpovědi na požadavek na otevření souboru.

Povinné vs. nezávazné řízení přístupu

Mandatory vs. Discretionary Access Control

Bezpečnostní mechanismy pro řízení přístupu se dají rozdělit do dvou skupin:

- **Nezávazné řízení přístupu (DAC):**

- Uživatelé mohou sami rozhodovat o přístupu (např. měnit práva ostatních uživatelů ke svým souborům)
- Mohou delegovat svá přístupová práva ostatním uživatelům

- **Povinné řízení přístupu (MAC)**

- Je vynucována administrátorem definovaná politika
- Uživatelé ji nemohou měnit (pokud to politika explicitně nepovoluje)
- Může zabránit nedůvěryhodným aplikacím běžícím s právy uživatele v páčání škody.

Přetečení bufferu (buffer overflow)

- Jedna z nejčastějších chyb programátorů v C, často i v C++
- Skoro vždy se dá nějak zneužít
- Hodně zajímavé (z hlediska útočníků) je přetečení bufferu na zásobníku (lokální proměnná)
 - Tzv. stack smashing attack
 - Zneužitelnost chyby je dnes na velkých systémech (servery, PC) částečně eliminována (viz dále)
 - Problém je ale...

IoT

~~Internet~~ of Things
Insecurity

Co je to “frame pointer”?

- A) Ukazatel na začátek stránky paměti, do které se přistupuje
- B) Ukazatel na začátek (konec) lokálních proměnných právě vykonávané funkce
- C) Ukazatel na začátek kódového segmentu aktuálního programu

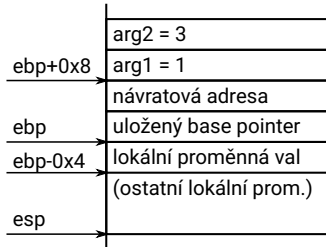
Zásobník – jak tam jsou uložena data?

Záleží, jestli je kód přeložen s nebo bez „frame pointeru“

```
int func(int arg1, int arg2)
{
    volatile int val=arg1+arg2;
    return val;
}

void main()
{
    func(1, 3);
}
```

Zásobník uvnitř func()



S ukazatelem rámce (gcc -fno-omit-frame-pointer)

```
func:
    push    %ebp                ; ulož bp na zásobník
    mov     %esp,%ebp          ; nastav ebp jako na obr.
    sub     $0x10,%esp         ; nastav esp jako na obr.
    mov     0xc(%ebp),%eax      ; načti arg2
    add     0x8(%ebp),%eax      ; přičti k arg1
    mov     %eax,-0x4(%ebp)     ; ulož do prom val
    mov     -0x4(%ebp),%eax     ; zkopíruj val do eax
    leave   ; obnov ebp
    ret                        ; vrať se do main
```

```
main:
    push    %ebp
    mov     %esp,%ebp
    push    $0x3                ; ulož parametry funkce
    push    $0x1                ; ... na zásobník
    call    500 <func>          ; zavolej func
    pop     %eax
    pop     %edx
    leave
    ret
```

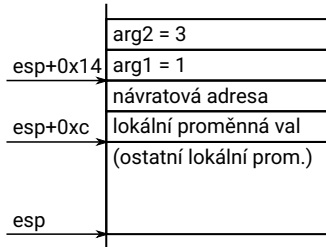
Zásobník – jak tam jsou uložena data?

Záleží, jestli je kód přeložen s nebo bez „frame pointeru“

```
int func(int arg1, int arg2)
{
    volatile int val=arg1+arg2;
    return val;
}

void main()
{
    func(1, 3);
}
```

Zásobník uvnitř func()



Bez ukazatele rámce (gcc -fomit-frame-pointer)

```
func:
    sub    $0x10,%esp        ; nastav esp jako na obr.
    mov    0x18(%esp),%eax    ; načti arg2
    add    0x14(%esp),%eax    ; přičti k arg1
    mov    %eax,0xc(%esp)     ; ulož do prom. val
    mov    0xc(%esp),%eax     ; zkopíruj val do eax
    add    $0x10,%esp        ; posuň esp k návř. adr.
    ret                                ; vrať se do main
```

```
main:
    push   $0x3              ; ulož parametry funkce
    push   $0x1              ; ... na zásobník
    call   500 <func>        ; zavolej func
    pop    %eax
    pop    %edx
    ret
```

Využití frame pointeru

- Pokud dostanu adresu rámce na zásobníku, můžu snadno vypsat všechny rámce (např. tzv. backtrace v debuggeru nebo BRUTE).
- Pokud frame pointer není ukládán, debugger musí informaci o rámcích získat někde jinde. Např. z ladicích informací vygenerovaných překladačem.

Přetečení zásobníku

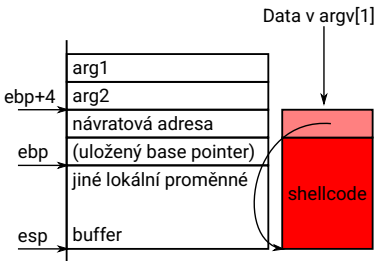
- Programátor zapomene zkontrolovat velikost proměnných na zásobníku
- Uživatel může předat programu víc dat, než je velikost proměnné na zásobníku
- To může způsobit přepsání dalších lokálních proměnných, návratové adresy, parametrů, ...
- Program pak většinou „spadne“ (segmentation fault)
- Nebo toho můžeme zneužít a donutit program, aby dělal to, co chceme my.

Zneužití přetečení zásobníku

Chceme, aby cizí program dělal to, co chceme my a ne to, co chtěl jeho autor.

```
int main(int argc, char *argv[])
{
    char buffer[10];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Spuštění: `./prog "$(cat shellcode)"`



Shellcode

- Typickým cílem útočníka je spuštění shellu, tj. chtěl by spustit následující kód:

```
dup2(socket, 0);
dup2(socket, 1);
execve("/bin/sh", NULL, NULL);
```

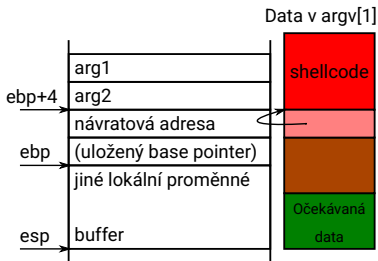
- Shellcode nemusí jen spouštět shell, celý útok (změna obrázku na serveru :) může být součástí zaslaného kódu.
- Když se shellcode nevejde do proměnné `buffer`, je možné přepsat i rámce volajících funkcí – tam může být víc místa.
- To samé se může hodit, pokud kód kontroluje, že se v `bufferu` objeví "očekávaná data".

Zneužití přetečení zásobníku

Chceme, aby cizí program dělal to, co chceme my a ne to, co chtěl jeho autor.

```
int main(int argc, char *argv[])
{
    char buffer[10];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Spuštění: `./prog "$ (cat shellcode)"`



Shellcode

- Typickým cílem útočníka je spuštění shellu, tj. chtěl by spustit následující kód:

```
dup2(socket, 0);
dup2(socket, 1);
execve("/bin/sh", NULL, NULL);
```

- Shellcode nemusí jen spouštět shell, celý útok (změna obrázku na serveru :) může být součástí zaslaného kódu.
- Když se shellcode nevejde do proměnné `buffer`, je možné přepsat i rámce volajících funkcí – tam může být víc místa.
- To samé se může hodit, pokud kód kontroluje, že se v `bufferu` objeví "očekávaná data".

Omezení shellcode

- Strojový kód většinou nesmí obsahovat binární nuly, protože funkce jako `strcpy` předpokládají řetězec ukončený nulou a nezpracovaly by všechna data.
- **Příklad:** Instrukce `mov $1,%eax` ukládá do `eax` hodnotu 1. Ve svém kódování má tři nulové byty, protože 32bitová hodnota 1 tj. `0x00000001` je součástí instrukce. Tuto instrukci můžeme nahradit vynulováním instrukcí `xor` a zvětšením o jedna:

```
B8 01000000 mov $1,%eax
```

nahradit za:

```
33C0          xor %eax,%eax
```

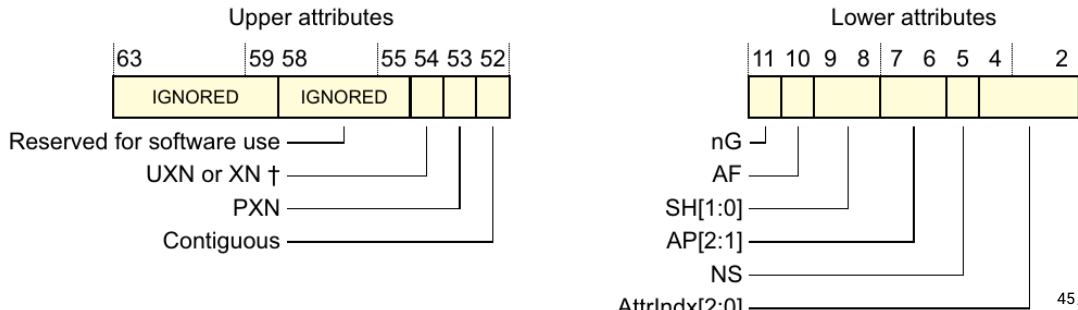
```
40           inc %eax
```

Nespustitelný zásobník

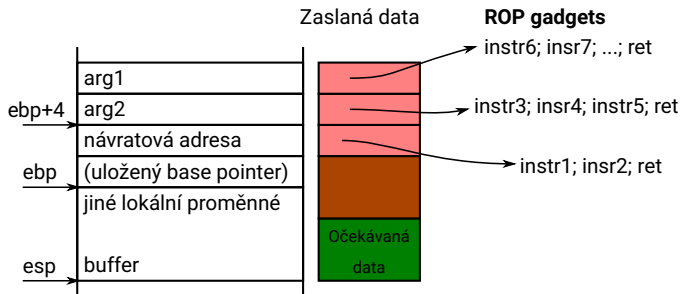
- Intel zavedl od PAE stránkování (PAE a x86_64) XD bit (eXecute-Disable)
 - Při pokusu o vykonání kódu ze stránky s XD=1 dojde k výjimce, která způsobí, že OS daný proces ukončí (segfault).
 - OS alokuje paměť pro zásobník s XD=1.

X D	Reserved	Address of 4KB page frame	Ign.	G	P A T	D	A	P C D	P W T	U /S	R /W	<u>1</u>	PTE: 4KB page
--------	----------	---------------------------	------	---	-------------	---	---	-------------	-------------	---------	---------	----------	---------------------

- ARM používá UXN/XN bit (Unprivileged eXecute Never)



Return-oriented programming (ROP)



- Když nejde spustit (shell) kód na zásobníku, může se útočník pokusit spustit kód, který už v programu je.
- Že by se v programu nacházel přesně ten kód, který útočník potřebuje, je nepravděpodobné.
- Jinde je ale spousta „zajímavého“ kódu – např. v knihovně `libc` najdeme kód, který vyvolává všechna možná systémová volání.
- ROP = Útočník neposílá přímo shellcode, ale sekvenci návratových adres, které způsobí postupné vykonání kousků kódu (ROP gadgets), který se nachází v jiných částech programu.

ROP – pokračování

■ Existují ROP překladače

- Předloží se jimi program a knihovny, na které chceme útočit (např. webový server a knihovny z populární Linuxové distribuce)
- kompilátor přeloží zdrojový kód v C do ROP programu (sekvence návratových adres, které je potřeba uložit na zásobník).

Náhodné rozložení adresního prostoru

Address space layout randomization (ASLR)

- Pro většinu typů útoků se zásobníkem je potřeba znát adresy, na které lze „skákat“ instrukcí ret.
- Pokud útočník neumí adresy zjistit, jsou útoky těžké či nemožné.
- Sdílené knihovny jsou zkompileovány tak, že je lze nahrát a spustit z libovolné adresy (position independent code – PIC)
 - Linkování se provádí až při spuštění, takže je možné je umístit při každém startu na jinou adresu.
- I program lze přeložit jako PIC (značí se PIE – position independent executable) a zásobník také nemusí být na pevné adrese.
- Zkuste si v GNU/Linuxu spustit: `watch -d cat /proc/self/maps`. Uvidíte, že při každém spuštění příkazu `cat` jsou adresy jiné.

Když ASLR nestačí

- Jádra OS nemohou používat tak intenzivní ASLR jako uživatelský prostor.
- Linux používá náhodnou adresu zásobníků v jádře, ale adresa kódu se zvolí náhodně jen při bootu, pak zůstává stejná.
- Možná řešení: stack protector, stack canary, Retguard (OpenBSD)
- Retguard:
 - Při vstupu do funkce zakóduje návratovou adresu
 - ESP se dá považovat za náhodné – je těžké ho uhádnout
 - Před návratem se návratová adresa obnoví XORem
 - Pokud útočník přepsal návratovou adresu, obnovou se jeho adresa znehodnotí ⇒ systém „spadne“

Retguard – příklad:

main:

```
push    $0x3
push    $0x1
call    500 <func>
pop     %eax
pop     %edx
ret
```

func:

```
xor     (%esp),%esp ; zakoduj
sub     $0x10,%esp
mov     0x18(%esp),%eax
add     0x14(%esp),%eax
mov     %eax,0xc(%esp)
mov     0xc(%esp),%eax
add     $0x10,%esp
xor     (%esp),%esp ; obnov
ret
```

Závěr

- Bezpečnost je důležitým aspektem každého počítačového systému
- V budoucnosti bude její důležitost narůstat
- Systémy (nejen operační) jsou tak bezpečné, jak bezpečný je nejslabší článek
 - I ta nejméně důležitá knihovna používaná vaším programem může obsahovat kritickou zranitelnost
 - I operační systém obsahuje mnoho komponent, které nepoužíváte, ale útočníkům pomohou
- Útočníci jsou velmi kreativní a vynalézaví lidé
- Pokud se jim chcete bránit, musíte umět myslet jako oni

Reference

- Využili jsme některé materiály licencované pod CC BY 3.0 „Courtesy of Gernot Heiser, UNSW Sydney“.