

Jak se smaží zásobník

Radoslav Bodó <bodik@civ.zcu.cz>

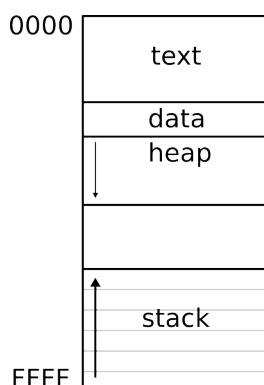
Hitem současných let je a bude web, webové aplikace, SOA, XML .. Tyto nové, úžasné technologie však zastínili staré bojové pole, na kterém však stále probíhá bitva o přístup a ochranu paměti. V tomto článku se pokusíme shrnout a prezentovat techniky útoků a obran na aplikace a operační systémy typu cokoliv-overflow. Článek vychází převážně z publikací konference BlackHat 2007/2008/2009.

Klíčová slova: buffer overflow, call stack, fandango on core, AlephOne, memory protection, /GS, SafeSEH, DEP, PaX, W^X, ASLR, GrSecurity

Hned v úvodu bych rád upozornil čtenáře, že všechny následující materiál byl více či méně převzat z uvedené literatury a neobsahuje žádný vlastní výzkum. Článek by měl posloužit zájemcům jako rozcestník pro danou problematiku a případně aktualizaci znalostí.

O tématu stack buffer overflow bylo napsáno již velmi mnoho materiálu, proto zde uvedeme pouze krátkou rekapitulaci. Následující část osvětlí základní stavební prvky této problematiky a důsledky zneužití chyb tohoto typu. Předpokládáme, že cílem našeho snažení je získání kontroly nad procesem pomocí zneužití chyby v programu. Případné zájemce o hlubší průzkum úvodní problematiky viz [Aleph1].

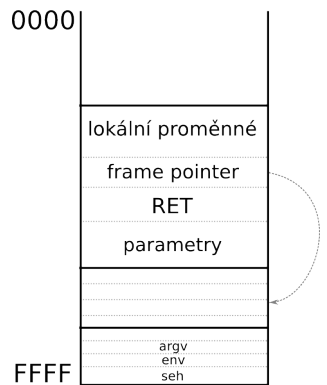
Vyrovnávací paměť, přetečení a zásobník



Obrázek 1: Rozložení adresního prostoru procesu

Buffer je spojitý blok paměti, který obsahuje pole prvků jednoho datového typu, ne vždy nutně pole znaků. O **přetečení/overflow** bufferu hovoříme v případě, že při práci s výše zmíněným bufferem dojde k zápisu do paměti za jeho hranice. Rozložení částí programu v adresním prostoru (u počítačů s von Neumanovou architekturou) procesu ukazuje následující obrázek. Text představuje kód programu, data jeho statická data, heap jeho dynamicky alokovaná data a stack jeho zásobník.

V dnešních počítačových systémech jsou programy strukturovány do procedur a funkcí které se navzájem volají. Pro udržení informací o průběhu programu, tj. zanořování jednotlivých funkcí, používají programy k uložení potřebných dat a metadat paměťový prostor nazývaný **stack**. Stack je používán pro: předávání parametrů, ukládání návratových adres (RET/%EIP), uložení pointerů potřebných pro zřetězování/řízení stacku a práci s jeho daty (SFP/%EBP), uložení lokálních proměnných funkcí, na Windows navíc pro ukládání adres obsluh výjimek.



Obrázek 2: Data na zásobníku

Volání funkce programu má zpravidla 3 části:

- *prologu* – části, kde probíhá příprava parametrů volání funkce, uložení adresy na které má program pokračovat po návratu z funkce, zapamatování ukazatele na aktuální stack frame a vytvoření prostoru pro lokální proměnné volané funkce,
- *těla funkce* – samotného těla funkce,
- *epilogu* – části, kde se řízení programu vrací do volající funkce pomocí uložené adresy RET a obnovení stacku volající funkce.

Protože v současných počítačích není oddělen adresní prostor řídicích dat na stacku (SFP, RET) a lokálních dat programu/funkce, může za určitých okolností dojít k přepsání řídicích dat (adresy RET) ...

```
-----
void function(char *str) {
    char buffer[32];

    strcpy(buffer,str);
}

void main(int argc, char *argv[]) {
    function(argv[1]);
}
-----
```

Obrázek 3: Ukázka programu s chybou buffer overflow

Ve výše uvedeném příkladu je nejjednodušší případ pro stack overflow. Pokud bude jako parametr programu předán řetězec větší než 32 znaků, dojde na stacku k přepsání návratové adresy (i frame pointeru) a program, při návratu z funkce, skončí skokem do neznáma ... Útočník se může pokusit vložit takový řetězec, který do RET vloží nějakou smysluplnou hodnotu, aby tak program pokračoval na zvoleném místě a udělal něco pro útočníka .. například spustil *execve(/bin/sh)*, tj. nahradil současný proces shellem, aby mohl útočník spouštět další příkazy. Pokud by byl výše uvedený program označen jako *setuid*, získal by výsledný shell příslušná práva.

Vkládanému řetězci se zpravidla říká **shellkód** a musí splňovat několik parametrů v závislosti na použití:

- musí být zapsán správně pomocí instrukcí pro daný procesor,
- neměl by obsahovat žádné absolutní adresy (PIC),
- neměl by obsahovat znaky 0x0, které neprojdou přes funkce, které zpracovávají řetězce.

```
-----
int main(void) {
    char buf[] = "Hello world!\n";
    write(1, buf, sizeof(buf));
    exit(0);
}
-----
```

Obrázek 4: Předloha pro shellkód

```
-----
hAAAAX5AAAAHPPPPPPPh0B20X5Tc80Ph0504X5GZBXPh445AX5XXZaPhAD00X5wxuPTYII19hA000
X5sOkkPTYII19h0000X5cDi3PTY19I19I19I19h0000X50000Ph0A0AX50yuRPTY19I19I19I19h0000
```

```
X5w100PTYIII19h0A00X53sOkPTYI19h0000X50cDiPTYI19I19hA000X5R100PTYIII19h00A0X500y
uPTYI19I19h0000X50w40PTYII19I19h0600X5u800PTYIII19h0046X53By9PTY19I19I19h0000X50
VfuPTYI19I19h0000X5LC00PTYIII19h0060X5u79xPTY19I19I19h0000X5000FPTY19I19h2005
X59DLZPTYI19h0000X500FuPTYI19I19h0010X5DLZ0PTYII19h0006X50Fu9PTY19I19I19h0000
X5LW00PTYIII19h0D20X5Lx9DPTY19h0000X5000kPhA0A0X5ecV0PTYI19I19h0B0AX5FXLRPTY19h5
550X5ZZZPePTYI19jã
```

Obrázek 5: Výsledný ascii shellkód s alpha loaderem [Shellforge]

V reálných případech není situace tak jednoduchá jako v našem příkladu. Pro úspěšné zneužití potřebuje útočník: jakkoli nahrát vlastní kód do paměti programu, případně najít adresu kde se užitečný kód již nachází a přepsat zásobník tak, aby RET začal ukazovat do místa kam byl shellkód vložen.

Nicméně zjistit správnou adresu nemusí být vždy jednoduché ani pro jednoduché programy, nehledě na to, že různé kompilátory používají různá zarovnávání a optimalizace, takže na různých počítačích může být stack frame zneužívané funkce ve velmi odlišných místech. Tento fakt se dá obejít několika způsoby:

- zvětšením shellkódu pomocí několika instrukcí NOP, tím zvětšíme rozsah paměti, kam může RET ukazovat,
- pokud je buffer příliš malý a můžeme kontrolovat proměnné prostředí, můžeme shellkód vložit do nich, protože jsou vždy na začátku stacku a mohou být *libovolně* veliké,
- pokud ani na to není prostor je možné použít návrat na nějaký registrový skok (jmp %reg),
- můžeme skákat na adresu funkce z některé sdílené knihovny (ret2lib),
- případně do textového segmentu samotného programu.

Některé zajímavé příklady z praxe ukazují, že tento typ chyb je i přestože velmi starý, stále rozšířený a mezi útočníky oblíbený:

- [Morris worm](#) – pro šíření zneužíval chybu v unixovém démonu finger (1988)
- [Slammer worm](#) – se šířil pomocí chyby v [Microsoft's](#) SQL serveru (2003)
- [Blaster worm](#) – šířil se díky chybě v Microsoft [DCOM \(2003\)](#)
- [Witty worm](#) – pro šíření zneužíval chybu ve firewallu [Internet Security Systems](#) (2004)
- Conficker A – pro šíření využíval chybu v RPC (MS08-067; 2008)

Obrany a jejich obcházení

Proti výše uvedené chybě bylo vynalezeno několik více či méně účinných ochran, všechny však byly poraženy velmi záhy po svém uvedení.

Odstranit příčinu

První možností která se samozřejmě nabízí je naučit programátory nedělat chyby – což je nemožné. Druhou je pokusit se používat zabezpečené varianty problematických funkcí (*libsafe*), buffer overflow však nenastává pouze při práci s řetězci. Dále se nabízí statická kontrola kódu, ale ani *splint* ani *valgrind* není všemocný, nehledě na to že by jejich použití muselo být běžnou součástí vývojového procesu.

Existují i rozšíření překladačů (gcc FORTIFY_SOURCE), které kontrolují práci s buffer při překladu, tato ochrana funguje pouze pro některé programové konstrukce. Též je možné kontrolovat programy za běhu (bounds checking), tato technika je však výkonnostně náročná.

Také bychom mohli používat pouze jazyky, které nepodporují přímé ukazatele do paměti (Java, Python, Ruby), ale kdo by chtěl psát operační systém v Jave.

NX/XD/XN/DEP – nespustitelné stránky

Myšlenka jednoduchá, rozšířit metadata stránek o další bit (RW > RWX) a označit paměťové stránky stacku jako

nespustitelné. Pokud do takové stránky umístí útočník svůj kód a pokusí se jej vykonat procesor mu to nedovolí. Tato vlastnost je do procesorů implementována od zhruba od roku 2004, ale i před tím byly projekty (PaX, ExecShield, W^X) které to dokázali softwarově emulovat ...

Implementace této ochrany vyžaduje jak podporu v operačním systému, tak i ve spouštěném programu. Ne všechny jsou ale schopné s tímto korektně fungovat. Např: Java, Lisp, Xserver, ... používají stack pro dynamické vytváření a spouštění kódu (trampolíny, ...) a proto pro ně musí být vytvořeny výjimky.

Softwarové verze této ochrany lze někdy vypnout odskokem (pomocí RET) přes volání *mprotect()*, které s nastavením zabezpečení stránek manipuluje.

Windows XP mají podporu NX od SP2 s defaultní politikou *OptIn* (stejně tak Vista), která říká že DEP je zapínáno na požádání, Windows 2003, 2008 mají politiku *OptOut*. Na 64bitových Windows jsou již všechny programy se zapnutým DEP, který nejde vypnout, ale Internet Explorer zde zůstává stále 32b proces a podléhá výše uvedeným politikám a lze tuto ochranu vypnout [Browsers]. Stejně tak DEP nepodporoval Firefox 2.

Hlavním způsobem jak obejít tuto ochranu je však technika return-into-libc [ret2lib]. Pokud není možné vykonávat kód uložený na stacku, není nic jednoduššího než použít již existující kód, který je určen ke spuštění – kód sdílených knihoven nebo samotného programu. Oblíbenou technikou je tedy příprava parametru na stack a skok do funkce *system()* z knihovny libc. Příprava stacku pro tuto funkci může vyžadovat zvětšení parametru (např: `//////////./////////bin/sh`).

V případě nových procesorů x86_64 se předávání parametrů volání provádí nikoliv pamětí na stacku ale registry procesoru, toho je možné dosáhnout technikou *ret code chunking – gadgets* [Negral, Schacham2, Fritsch1]. Nejdříve odskočit na nějaké místo v paměti, kde se nacházejí instrukce typu *'pop %rdi; ret'* které natáhnou hodnotu ze stacku do registru a až poté odskočit do funkce *system()*. Takto je možné zřetěžit více potřebných instrukcí nebo funkcí.

ASLR – prostor plný náhody

Jak je vidět, jednu z hlavních rolí v buffer overflow hraje znalost adres kódu nebo dat, které útočník potřebuje, ať už se jedná o jím vložený shellkód, nebo adresy užitečných funkcí. Před ASLR byly všechny části programu, sdílené knihovny i stack mapovány na stejná místa ve virtuálním adresním prostoru procesu. Řešením by tedy mohlo být, změnit toto statické mapování tak, aby měl každý proces knihovny, kód, data i stack pokaždé na jiném místě v paměti. To by mělo znemožnit útočníkovi nalézt potřebné adresy. Jako první navrhl a implementoval tuto myšlenku projekt PaX v roce 2001 [PaX].

V Linuxu je od jádra verze 2.6.12 znáhodněna část virtuální adresy, 8 bitů pro x86_32 a 28 bitů na x86_64 [Fritsch1]. Windows podporují tuto ochranu od verze Vista, např. haldu znáhodňují v rozsahu 2MB. Implementace ASLR na Mac OS X je v plenkách, znáhodňují se adresy knihoven, ale už ne dynamického linkeru ze kterého lze pak informace získat [MacOS].

Bohužel, kód programu, musí být přizpůsoben k relokační a mnoho softwaru dodnes není (např. plugíny browserů). Útočník může vložit do stránky takový EMBED objekt, který způsobí natažení nerelokovatelného pluginu a potom jej zneužít za znalosti adresy kam se nahrál. Stejně tak implementace nemusí být bez chyby, v linuxovém jádře (< 2.6.20) nebyla znáhodněna adresa na kterou se mapovala knihovna *linux-gate.so* - knihovna, která poskytuje programu možnost volat jádro bez použití přerušení (které je pomalé). Tato chyba umožnila provádět return-into-libc i na systémech s ASLR. Navíc, není v linuxu defaultně znáhodněno mapování samotného programu.

Linuxových 8 bitů (pro x86_32) není dnes příliš a tak se dá adresa odhadovat i hrubou silou [Schacham1]. Volání *fork()* totiž zachovává náhodnost adresního prostoru a tak je zde prostor pro pokusy správnou adresu odhadnout. Proti tomuto útoku zavedl projekt GrSecurity+PaX znemožnění volání *fork()* pokud proces spadl vícekrát než je za časový úsek zdrávo, navíc používá větší míru znáhodnění adresy (podobně jako ExecShield).

Další možností [Browsers] je sprejování. Pokud může útočník alokovat dostatečné množství místa na shellkód (řádově megabajty až stovky megabajtů), může zvýšit své šance pro úspěšné odhadnutí potřebné adresy. Manipulace s proměnnými prostředím nebo používání jazyků jako JavaScript, Java, Flash tyto alokace dovolují. Na obrázku 6 je ukázka z exploitu pro Internet Explorer [Milw0rm].

```

-----
...
var shellcode = unescape("%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f...");
var block = unescape("%u0c0c%u0c0c");
var nops = unescape("%u9090%u9090%u9090");

while (block.length < 81920) block += block;
var memory = new Array();
var i=0;
for (;i<1000;i++) memory[i] += (block + nops + shellcode);

document.write("<iframe src=\"iframe.html\">");
</script>
</html>

<!-- iframe.html
<XML ID=I><X><C><![CDATA[
    <image
        SRC=http://&#3084;&#3084;.xxxxx.org
    >
]]></C></X></XML>
-----

```

Obrázek 6: Ukázka exploitu MS IE XML Parser Remote Buffer Overflow

Metodou částečných přepisů (partial overwrites), může útočník zachovat základ adresy, tu část ve které je znáhodnění, a přepisem 1 nebo 2 nejméně významných bitů přesto dosáhnout potřebných manipulací s pamětí na základě znalosti relativního umístění objektů v paměti.

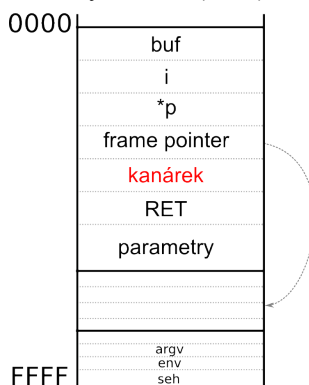
Pokud všechny výše uvedené metody selhávají může se útočník pokusit znáhodnění prostoru odhalit pomocí chyb formátování řetězců.

Zřejmě nejnovější technika [Fritsch2] zneužívá další chyby implementace ASLR v současném linuxovém jádře. Zde je náhodnost rozložení závislá na PID procesu a *case*. Při svém výzkumu Fritsch zjistil, že existuje časové okno veliké až 2 minuty, kdy dokáže útočník nasimulovat stejná, nebo velmi podobná, znáhodnění pro různé procesy za pomoci manipulace s PID, které dokáže ovládat pomoci volání *fork()*, *execve()* a *usleep()*. S pomocí získaných hodnot pak zjistit adresy potřebné pro úspěšné zneužití buffer overflow, to za použití několika málo pokusů (na platformě x86_64).

A nakonec Kanárek

V roce 1997 přišel Cristin Cowan s dalším nápadem (*StackGuard*), v prologu funkce umístit na stack mezi lokální proměnné a návratovou adresu nějakou hodnotu/kanárka/cookie a v epilogu, před skokem na RET, zkontrolovat, zda nebyla změněna. Pokud by došlo v průběhu funkce k útoku, byla by tato hodnota přepsána a útok odhalen [Moyer].

Zhruba 24 hodin po zveřejnění projektu se ukázalo, že statické hodnoty nejsou žádnou reálnou ochranou a buď se dají odhadnout hrubou silou, nebo z paměti přečíst či přepsat jejich předlohu. Postupem času byla technika této ochrany vylepšena na náhodný XORovaný kanárek, který se velmi špatně odhaduje, případně kanárek který se nedá kopírovat funkcemi pro práci s řetězci (tzv. terminator canary - NULL(0x00), CR (0x0d), LF (0x0a) a EOF (0xff)).



Obrázek 7: Ukázka zásobníku s kanárkem

Nicméně, původní implementace nechránila frame pointer, pomocí jehož přepisu (nebo přepisu jeho části) bylo dále možné manipulovat se stackem a lokálními proměnnými volající funkce (caller's frame pointer) [Richarte]. Později byl kanárek posunut (první Microsoft /GS, posléze i gcc) mezi frame pointer a lokální proměnné.

I přesto však zbývá prostor pro útok, útočník může stále přepisovat lokální proměnné, což v případě přepsání pointeru může vést k zápisu na libovolnou adresu v paměti podobně jako u heap overflow. Tímto způsobem se může útočník pokusit přepsat pointer na funkci, která je později za jeho pomoci volána nebo se může pokusit přepsat GOT/PLT [Prielom1], tabulku která drží informace o adresách relokovaných knihovních objektů a funkcí, které jsou použity ještě před či dokonce při kontrole kanárka a nebo obecně, dosáhnout zápisu na vybrané místo v paměti.

```
-----
/* sg1.c *
* specially crafted to feed your brain by gera@corest.com */
int func(char *msg) {
    char buf[80];
    strcpy(buf,msg);
    // toupper(buf); // just to give func() "some" sense
    strcpy(msg,buf);
}

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

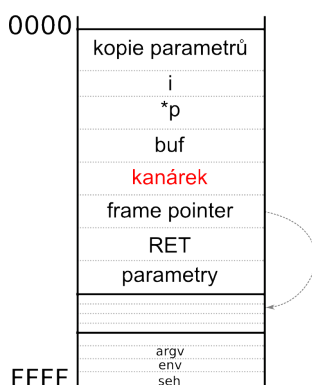
Obrázek 8: Zápis na vybrané místo v paměti [Richarte]

Na podobném principu pracuje zneužití handlerů výjimek. Na systémech Windows, jsou na stacku uloženy i adresy obsluhy výjimek (SEH), ty jsou sice uloženy až za kanárkem a měly by tedy být chráněny, ale pokud nastane ve funkci výjimka (např. při přetečení nějakého counteru s následným přístupem do nealokované paměti), dojde k použití této adresy ještě před kontrolou kanárka. Windows se snaží implementovat SafeSEH, tj. mít v paměti seznam bezpečných obsluh výjimek, ale opět je tato vlastnost zapnuta pouze v případě plné podpory DEP a SEH v programu a mnoho knihoven tuto ochranu dosud nepodporuje.

Kolem roku 2004/2005 byly kompilátory obohaceny o schopnost přeskládat lokální proměnné tak, aby byly zneužívané buffery uloženy před ostatní lokální proměnné a navíc doplnit stack o kopii parametrů funkce (projekt *ProPolice*). Pokud tedy dojde k přetečení některého z bufferů, neovlivní to lokální proměnné, ani parametry funkce a navíc dojde k přepsání kanárka, což je při návratu z funkce detekováno a proces je ukončen.

Práce a kanárkem, ale přidává do programů nezanedbatelnou režii, proto nebývalo jejich používání implicitně zapnuto a navíc jsou tímto způsobem chráněny pouze některé funkce, zpravidla ty které používají znakové buffery o délce větší než 4. Stále tedy zbývá dostatek funkcí, které chráněné nejsou a přesto se dají zneužít (CVE-2007-0038, CVE-2006-3747).

V neposlední řadě může, podobně jako u ASLR, dojít chyb formátovacích řetězců či nechráněných frame pointerů k vyzrazení hodnoty kanárka a tudíž eliminaci této ochrany. Snad poslední chybou do výčtu chybí, byla implementační část v GCC, kde je inicializační část kanárka závislá na statických hodnotách a čtení náhodných dat z */dev/random* bývalo z výkonnostních důvodů defaultně vypnuto [Fritsch1]. Distribuce Debian zapnula tuto ochranu ve verzi 5.0/Lenny.



Obrázek 9: Ukázka stacku při použití ProPolice

Alternativou ke StackGuardu je StackShield, ten klonuje návratovou adresu do místa v paměti které se nedá přepsat, a před návratem z funkce jí vrací na původní místo bez porovnání. Tato ochrana je celkem silná, nicméně stále neposkytuje ochranu proti manipulaci s lokálními proměnnými, argumenty nadřazené funkce a přepis GOT, navíc nedetekuje pokusy o útok, pouze částečně eliminuje jejich dopad [Prielom1].

Závěr

V tomto článku jsem se pokusil shrnout metody, které byly navrženy a implementovány jako obrana proti chybám typu stack buffer overflow. Na většinu z nich byl nalezen protiútok velmi záhy po uvedení ochran. Podle výzkumu který provedl Hagen Fritsch [Fritsch1], jsou ochrany zásobníků pomocí kanárků zatím nejúčinnější, ale hlavně v kombinaci s ostatními technikami ASRL a NX.

Mohlo by se zdát, že hlavní chybou je použití von Neumanovy architektury, ale nejnovější výzkumy ukazují, že se dají nalézt techniky pro permanentní vkládání a spouštění kódu i na platformách založených na Harvardské architektuře [Harvard].

Literatura a odkazy:

- [Aleph1] *Aleph One: Smashing The Stack For Fun And Profit*
Phrack 49
- [Browsers] *Alexander Sotirov, Mark Dowd: Bypassing Browser Memory*
Black Hat 2008
- [ret2lib] *Solar Designer: Getting around non-executable stack (and fix)*
- [Negral] *Nergal: The advanced return-into-lib(c) exploits: PaX case study*
Phrack 58
- [Fritsch1] *Hagen Fritsch: Stack Smashing as of Today: A State-of-the-Art Overview on Buffer Overflow*
Protections on linux_x86_64
Black Hat 2009
- [Fritsch2] *Hagen Fritsch: Bypassing ASLR by predicting a process randomization*
Black Hat 2009
- [Schacham1] *Hovav Shacham et al.: On the Effectiveness of AddressSpace Randomization*
<http://www.cs.jhu.edu/~rubin/courses/fall04/asrandom.pdf>
- [Schacham2] *Hovav Shacham: The Geometry of Innocent Flesh on the Bone: Return-into-libcwithout function calls (on the x86)*
<http://www.cs.jhu.edu/~rubin/courses/fall04/asrandom.pdf>
- [PaX] Homepage of The PaX Team
<http://pax.grsecurity.net/>
- [MacOS] *Charlie Miller, Vincenzo Iozzo: Fun and Games with Mac OS X and iPhone Payloads*
Black Hat 2009
- [Linux] *M. Dobšíček, R. Ballner: Linux bezpečnost a exploit*
Kopp, 2004

- [Herout] *Pavel Herout: Učebnice jazyka C*
Kopp, 1994
- [Moyer] *Shawn Moyer: (un)Smashing the stack*
Black Hat 2007
- [Richarte] *Gerardo Richarte: [Four different tricks to bypass StackShield and StackGuard protection](http://www.coresecurity.com/files/attachments/StackguardPaper.pdf)*
www.coresecurity.com/files/attachments/StackguardPaper.pdf
- [Harvard] *Aurelien Francillon, Claude Castellucia: Code injection attacks on Harvard-Architecture devices*
<http://planete.inrialpes.fr/~ccastel/PAPERS/CCS08.pdf>
- [Prielom1] *wilder: exploitovanie cez plt (procedure linkage table) a got (global offset table)*
<http://www.hysteria.sk/prielom/15/#7>
- [Shellforge] *ShellForge*
<http://www.secdev.org/projects/shellforge>
- [Milw0rm] *[MS Internet Explorer XML Parsing Remote Buffer Overflow Exploit](http://www.milw0rm.com/exploits/7410)*
<http://www.milw0rm.com/exploits/7410>