Mijn eerste exploit

Herbert Bos

November 1, 2005

Als we een programma beschouwen als een reeks instructies in het geheugen van een PC, dan bestaat die reeks zelf weer uit een verzameling functies die elkaar over en weer aanroepen. Wanneer functie f functie g aanroept, dan wordt gesprongen naar de instructies die bij g horen. Als g is uitgevoerd wordt teruggesprongen naar de instructie in f direct volgend op de instructie die de aanroep deed. Er wordt daarom door de computer bijgehouden welke instructie dat is, door middel van een *stack* (stapel). Beschouw bijvoorbeeld het C programma van Figuur 1.

Zoals alle C programma's begint deze code in de funtie `main()`. Vandaaruit wordt een functie `kwetsbare_functie()` aangeroepen. In deze functie wordt data gelezen in een buffer genaamd `buf`. Hiervoor wordt de functie `lees_in_buf()` gebruikt. Normaal zou deze functie wellicht data van het netwerk lezen, maar voor het gemak lezen we nu uit een bestand.

De stack bevindt zich boven in het geheugen en groeit naar beneden. Figuur 2 illustreert de stack zoals die er uitziet net na aanroep van functie `kwetsbare_functie()`. Het eerste wat gebeurt is dat het *returnadres* op de stack wordt gezet. Als de functie klaar is en het `return` commando in regel 20 wordt uitgevoerd, dan wordt het returnadres op de stack gebruikt om naartoe te springen. Normaliter wordt dan gesprongen worden naar de machinecode die hoort bij regel 25 in `main()`.

Andere dingen die op de stack worden gezet, zijn de basepointer en de lokale variabelen. De basepointer is niet van belang voor deze discussie en wordt buiten beschouwing gelaten. De lokale variabele `buf[]` is echter de spil waarom de exploit draait. De buffer is 36 bytes groot, en er wordt op de stack dan ook keurig ruimte gereserveeerd voor deze 36 bytes. Helaas verzuimt de functie `lees_in_buf()` bij het lezen om rekening te houden met de grootte van buffer `buf`. Als meer dan 36 bytes worden gelezen dan stroomt de buffer over en wordt geschreven over de velden die onder `buf` liggen op de stack (d.w.z, op hogere adressen, omdat de stack van boven naar beneden groeit). Met andere woorden, over `basepointer` en `returnadres`!

Stel dat we weten dat het returnadres zich bevindt op adres 0xbffff60c (decimaal 3221222924). Het begin van `buf[]` bevindt zich dan op adres 0xbffff60c - 4 - 36 = 0xbffff5e4 (3221222924). Wat we nu kunnen doen is een programma van precies 44 bytes laden in `buf`. Het programma begint op op 0xbffff5e4 (dat wil zeggen, daar komt de eerst instructie van het programma). In de

```
 1 // Functie die een file leest in een buffer.  Bij een webserver lees
 2 // je niet van file maar bijvoorbeeld van een netwerkverbinding.  In
 3 // dat geval zult u deze functie moeten vervangen door iets dat van
 4 // het netwerk leest.
 5 void lees_in_buf (char *buf)
 6 {
 7   int fd = open ("hello.exploit", O_RDONLY); // open de file
 8
 9   // Lees data in buffer 'buf'. Er worden maximaal 64 bytes gelezen.
10   // Maar de buffer is maar 36 bytes lang. Als de file langer is dan
11   // 36 bytes (zoals bij 'hello.exploit), dan stroomt de buffer over.
12     read (fd, (void*)buf, 64);
13     close (fd);
14 }
15 int kwetsbare_functie () // deze functie bevat een kwetsvare buffer
16 {
17   char buf[36];     // deze buffer gaan we overstromen
18   lees_in_buf(buf); // m.b.v. deze functie
19   printf ("klaar met lezen\n");
20   return 0;
21 }
22 int main ()
23 {
24   kwetsbare_functie (); // roep de functie aan
25   return 1;
26 }
```

Figure 1: Het 'server' programma met een beveiligingslek

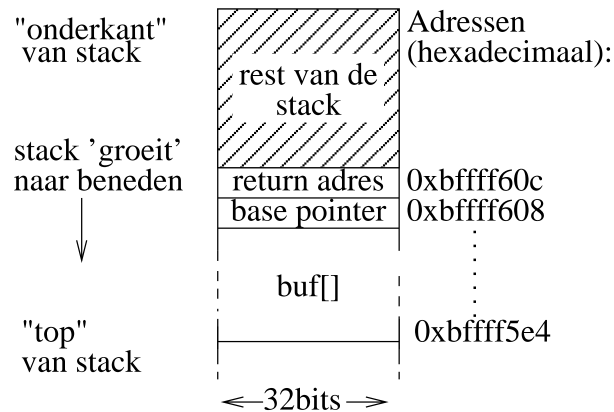

Figure 2: De stack bij een functieaanroep: array `buf[]` zit onder het returnadres

```
[SECTION .text]
global _start
_start:
        jmp short ender
        starter:
        xor eax, eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx
        mov al, 4       ;syscall write
        mov bl, 1       ;stdout is 1
        pop ecx         ;get the address of the string from the stack
        mov dl, 5       ;length of the string
        int 0x80
        xor eax, eax
        mov al, 1       ;exit the shellcode
        xor ebx,ebx
        int 0x80
        ender:
        call starter    ;put the address of the string on the stack
        db 'hello'
```

Figure 3: 'Hello' programma in assembly

laatste 4 bytes zetten we dan het *adres* waarop het programma begint, dus 0xbffff5e4. Deze 4 bytes overschrijven dan precies het return adres van functie `kwetsbare_functie()`. Met andere woorden, we schrijven een klein programmaatje en sturen dit naar het programma van Figuur 1 en tegelijkertijd overschrijven we het returnadres zodat er naar dit programma gesprongen wordt. Op het moment dat de `return` instructie wordt uitgevoerd in regel 20 in Figuur 1, wordt dan niet meer gesprongen naar de instructie die behoort bij regel 26, maar naar onze eigen instructie op adres 0xbffff5e4.

Als triviaal voorbeeld beschouwen we een programmaatje dat "hello" afdrukt op het scherm. Maar wacht, we kunnen natuurlijk niet zomaar een C programma sturen! In plaats daarvan moeten we een programmaatje in machinetaal genereren dat direct kan worden uitgevoerd. We noemen zulke programmaatjes die ons controle geven over de aan te vallen machine 'shellcode'. Deze code wordt meestal geschreven in assembly. Assembly is een stuk ingewikkelder in het gebruik dan C, maar wie een beetje assembly leert kan al snel heel elegante shellcode schrijven.

Het 'hello' programmaatje in assembly staat weergegeven in Figuur . De details zijn niet belangrijk, enkel de functionaliteit: het drukt 'hello' af op het scherm. Vervolgens halen we het programmaatje door een assembler en wat we overhouden is een binair blokje data, bestaande uit de instructies in machine taal. Voor het programma van Figuur is de binaire vertaling weergegeven als hexadecimale getallen bijvoorbeeld:

```
eb 19 31 c0 31 db 31 d2 31 c9 b0 04 b3 01 59 b2 05 cd 80 31 c0 b0
01 31 db cd 80 e8 e2 ff ff ff 68 65 6c 6c 6f
```

Dit moeten we nu in de variable `buf` plaatsen. Maar eerst moeten we het nog uitbreiden, want in deze code zit nog niet het adres 0xbffff5e4 verweven waarmee we het returnadres op de stack willen overschrijven. Dit plakken we er

eenvoudig achteraan. Als we het aantal bytes tellen in bovenstaande shellcode dan komen we op 37. We moeten in totaal 44 bytes hebben, waarvan de laatste 4 bytes gevormd worden door het adres. De tussenliggende 3 bytes maken we voor het gemak nul. De uiteindelijk shellcode is nu:

```
eb 19 31 c0 31 db 31 d2 31 c9 b0 04 b3 01 59 b2 05 cd 80 31 c0 b0
01 31 db cd 80 e8 e2 ff ff ff 68 65 6c 6c 6f 00 00 00 e4 f5 ff bf
```

We zien dat de oorspronkelijke shellcode nu gevolgd wordt door 3 nul-bytes en een 4 bytes adres. Misschien vraagt u zich af waarom het adres in omgekeerde volgorde staat opgeschreven: e4 f5 ff bf in plaats van bf ff f5 e4. Daar zit verder niets achter. Het is nu eenmaal de wijze waarop in een Pentium PC een 32-bits getal wordt gepresenteerd. Deze omgekeerde volgorde wordt wel Little Enidan genoemd. Andere machines maken gebruik van Big Endian en daar hoeven de bytes dus niet omgedraaid te worden.

Nu zijn we klaar. We kunnen bovenstaande shellcode in binaire vorm wegschrijven in een bestand `hello.exploit` en het 'server' programma starten met dit bestand als invoer. Het resultaat is dat 'hello' wordt afgedrukt op het scherm.

Gefeliciteerd, uw eerste exploit is klaar!