Aufgabenblatt #4: Buffer Overflows (15 Punkte)

4.1 Strcpy Vulnerabilities (4 Punkte)

Unter `/proj/i4syssec/ex/4/1` finden das Programm `vulnerable.c`, welches als Eingabe einen String erwartet und dann die Länge dieses Strings sowie den String selbst ausgibt.

```
student:~/ex/4/1# ./vulnerable this-is-a-string
(16) this-is-a-string
```

- Bringen Sie das Programm mit einer unerwarteten Eingabe zum Absturz. (0.5 Punkt)

  ```
  student:~/ex/4/1# ./vulnerable <your-evil-string>
  (...) <your-evil-string>
  Segmentation fault
  ```

- Erklären Sie wie es zum Absturz des Programms kommt. Welches ist die fehlerhafte Anweisung in `vulnerable.c`? Was passiert mit dem Return Instruction Pointer (RIP)? Zeichnen Sie bei Ihrer Erklärung den Stack auf (ASCII-Art genügt). (0.5 Punkte)
- Manipulieren Sie mit einer präparierten Eingabe den Programmfluss so, dass die Funktion `secret` ausgeführt wird.
  - Nutzen Sie dazu zunächst `objdump` oder `gdb` um die Adresse von `secret` zu bestimmen. (1 Punkt)
  - Schreiben Sie dann ein Skript (Perl, Python o.ä.) welches diese Adresse genügend oft aneinanderkettet. Achten Sie darauf die Adresse nicht im für Menschen lesbaren Text-Format auszugeben sondern binär im Little-Endian Format (d.h. die Adresse "0x61626364" würde als "dcba" ausgegeben werden). (1 Punkt)
  - Leiten Sie die Ausgabe Ihres Skriptes nun in die Eingabe von `vulnerable` um, so dass die Funktion `secret` ausgeführt wird. Nach Ausführung von `secret` darf das Programm abstürzen. (0.5 Punkte)

    ```
    student:~/ex/4/1/# ./vulnerable `./exploit`
    (...) <injection-vector>
    s3cr3t
    Segmentation fault
    ```

  - Erklären Sie kurz wie es zur Ausführung von `secret` kommt. Nehmen Sie dabei wieder Bezug auf den RIP und visualisieren Sie den Stack. (0.5 Punkte)

4.2 Shellcode (6 Punkte)

In der ersten Aufgabe haben Sie Buffer Overflows ausgenutzt um den Programmfluss zu manipulieren; Sie haben allerdings noch keinen eigenen Code injeziert und ausgeführt. In dieser Aufgabe sollen nun zunächst solche "Shellcodes" programmiert werden.

- Gehen Sie nach der Anleitung auf den Folien (oder `hackerwiki.org/index.php/Erstellen_eines_Shellcode`) vor um einen Nullbyte-freien Shellcode zu erstellen der Ihnen `/bin/sh` öffnet. Ihr Shellcode soll keine setuid-Funktionalität enthalten. (1 Punkt)

  ```
  student:~/ex/4/1/a# gcc -o shellcode shellcode.c ; ./shellcode
  sh-3.2$
  ```

  Nutzen Sie dazu folgendes Template:

  ```
  char shellcode[]= "put your shellcode here";

  int main(int argc, char **argv) {
          void (*shell)() = (void*)shellcode;
          shell();
  }
  ```

- Modifizieren Sie den Shellcode nun so, dass `touch hacked` ausgeführt wird statt `/bin/sh`. (3 Punkte)

  ```
  student:~/ex/4/1/b# ls
  shellcode.c
  student:~/ex/4/1/b# gcc -o shellcode shellcode.c ; ./shellcode
  student:~/ex/4/1/b# ls
  hacked shellcode shellcode.c
  ```

  Befreien Sie auch diesen Shellcode von Nullbytes. Nutzen Sie für Ihre Lösung keinen Generator, sondern schreiben Sie den Code selbst. Dokumentieren Sie daher Ihr Vorgehen und geben Sie Ihre Zwischenschritte, wie den Assembler-Code, mit an.

- Greifen Sie nun auf den Shellcode aus Aufgabenteil a zurück (`/bin/sh`) und befreien Sie ihn von numerischen Zeichen ('0' - '9', d.h. 0x30 - 0x39). (2 Punkte)

  Ihr Shellcode soll also folgende `sanitize` Funktion überstehen:

  ```
  void sanitize(char *str) {
          int i;
          for (i=0; i<strlen(str); i++)
                  if (str[i] >= '0' && str[i]<='9')
                          str[i]=0;
  }
  ```

  Geben Sie wieder Ihre Zwischenschritte, insbesondere den Assembler Code, mit an.

4.3 Address Space Layout Randomization (5 Punkte)

In dieser Aufgabe sollen die Kenntnisse aus Aufgabe 1 und 2 zusammengeführt werden, d.h. ein fehleranfälliges Programms soll so manipuliert werden, dass injezierter Shellcode ausgeführt wird.

- Alle modernen Betriebssysteme unterstützen ASLR um die Auswirkungen von Buffer Overflows einzudämmen. Erklären sie kurz wie ASLR funktioniert und warum damit einfache Exploits erfolgreich unterbunden werden. (0.5 Punkte)
- Auf `www.cs.tau.ac.il/tausec/lectures/Advanced_Buffer_Overflow_Methods.ppt` sind mehrere Methoden beschrieben um ASLR zu umgehen. Schauen Sie sich die Methode "ret2ret" genau an und erklären Sie mit eigenen Worten wie diese funktioniert. (1 Punkt)
- Unter `/ex/4/3/` finden Sie das Programm `vulnerable.c`. Exploiten Sie dieses trotz ASLR, d.h. bringen Sie Ihren Shellcode aus Aufgabe 1a zur Ausführung.
- Nutzen Sie `gdb` um zu analysieren wie genau die Stack-Frames von `main` und `function` während der Ausführung von `strcpy` aussehen. Visualisieren Sie dazu den Stack und tragen Sie alle relevanten Daten ein: Wo liegen die lokalen Variablen `buf`, `no` und `ptr`? Wohin zeigt `ptr`? Wohin zeigen die Register `esp`, `ebp` und `eip`? Wo liegt der Saved Frame Pointer, wo der Return Instruction Pointer? (1 Punkt)
- Nehmen Sie Ihre Visualisierung aus dem letzten Aufgabenteil und tragen Sie die Stackinhalte so ein, wie sie nach einem erfolgreichen Angriff auf die `strcpy` Schwachstelle aussehen müssten. Nutzen Sie hier die Methode "ret2ret". (0.5 Punkte)
- Führen Sie den "ret2ret"-Angriff den Sie oben theoretisch erarbeitet haben nun praktisch aus. D.h. erstellen Sie einen Angriffsvektor, der als Eingabe in `vulnerable` injeziert zur Ausführung von Shellcode führt. (2 Punkte)

Schreiben Sie dazu ein Programm oder Skript `exploit`, das sich wie folgt verhält:

```
student:~/ex/4/3# ./vulnerable `./exploit`
sh-3.2$
```

4.4 Zusatzaufgabe: ret2esp (5 Punkte)

Schauen Sie sich auf `www.cs.tau.ac.il/tausec/lectures/Advanced_Buffer_Overflow_Methods.ppt` eine weitere Methode zur Umgehung von ASLR an: "ret2esp". Exploiten Sie mit dieser Methode das Programm `vulnerable.c` unter `/ex/4/4/`. Gehen Sie wie bei Aufgabe 3 vor, d.h. erläutern Sie erst die Methode im Allgemeinen, beziehen Sie sie dann auf den Stack von `vulnerable` und führen Sie schließlich den Angriff aus.

Deadline: 28.01.2011