Rechnersicherheit, Reverse Engineering

Alexander Steen, Max Wisniewski

Übung 2

In- /Output

Input	yo	""	,	{	reverse	ing	}	is	==	fun
Output	49	175	11	29	42	25	41	27	80	19

Input	.	have	apple	/	cherry	&	keep	good	spirit	>	1.
Output	14	23	3	77	9	66	28	21	46	86	146

Die Eingaben haben wir ersten durch den Befehl *strings* erhalten und als wir ein paar Wörter hatte, fanden wir im *hexdump*, des Programms viele richtige Eingaben. Die Eingaben *1.* und *“”* haben wir durch *Socialengineering* erhalten.

Also kann man den gegebenen Output durch folgenden Anruf erzeugen:

```
printf "yo \"\" , { reverse ing } is == fun . have apple
/ cherry & keep good spirit > 1." | ./reverse_linux
```

Code

Das Programm vergleicht die Eingaben mit eingespeicherten Werten, die wir im Hexdump in einem Zusammenhängenden Speicherfeld gesehen haben (Das legt die Vermutung nahe, dass es sich um ein Array mit den Werten handelt).

Bei Eingabe wird für jedes Eingabewort die Funktion *next* aufgerufen, die wiederum *nextt* aufruft. In dieser wird entschieden, ob es sich um eine Zahl, ein Sonderzeichen, Schlüsselwort oder ein sonstiges Wort handelt.

Bei manchen Eingaben (z.B. ’$’ oder ’!’) wird der COde 17 zurückgegeben und das Programm terminiert (siehe main).

Funktion main

```
0x08048c2e <main+0>:    push   %ebp
0x08048c2f <main+1>:    mov    %esp,%ebp
0x08048c31 <main+3>:    and    $0xfffffff0,%esp
0x08048c34 <main+6>:    sub    $0x830,%esp
0x08048c3a <main+12>:   movl   $0x8048e34,(%esp)
0x08048c41 <main+19>:   call   0x804845c <puts@plt>
0x08048c46 <main+24>:   lea    0x14(%esp),%eax
0x08048c4a <main+28>:   mov    %eax,(%esp)
0x08048c4d <main+31>:   call   0x8048554 <init>
0x08048c52 <main+36>:   lea    0x14(%esp),%eax   <-----|
0x08048c56 <main+40>:   mov    %eax,(%esp)             |
0x08048c59 <main+43>:   call   0x8048c0f <next>        |
0x08048c5e <main+48>:   mov    %eax,0x82c(%esp)        | While-Schleife
0x08048c6a <main+60>:   mov    0x82c(%esp),%edx        | solange Eingabe
0x08048c71 <main+67>:   mov    %edx,0x4(%esp)          | nicht den Code 17
0x08048c75 <main+71>:   mov    %eax,(%esp)             | erzeugt
0x08048c78 <main+74>:   call   0x804844c <printf@plt> <|--(Hello World!)
0x08048c7d <main+79>:   cmpl   $0x11,0x82c(%esp)       |
0x08048c85 <main+87>:   je     0x8048c91 <main+99> ----|--|
0x08048c87 <main+89>:   cmpl   $0xf,0x82c(%esp)        |  | ansonsten
0x08048c8f <main+97>:   jne    0x8048c52 <main+36>   --|  | ende
0x08048c91 <main+99>:   leave                    <--------|
0x08048c92 <main+100>:  ret
```

Funktion init

Wir glauben, die Funktion *init* sorgt dafür das verscheidene Variablen initialisiert werden. So muss hier auch die Initialisierung des Arrays statt finden in der alle Werte stehen, die verschiedene Ausgaben erzeugen. (Das Array ist an der Stelle *0xd60* zu finden).

Der Pointer auf das Array wird so auf die Stelle im Speicher gelegt.

Funktion next

Unserer Meinung nach ist die Funktion next dafür verantwortlich, jedes Eingabewort (Mit Leerzeichen getrennt), dass in einer Eingabe (d.h. mit Carriage Return abgeschlossen) enthalten ist, entsprechend der Funktion nextt zu bearbeiten.

```
0x08048c0f <next+0>:    push   %ebp
0x08048c10 <next+1>:    mov    %esp,%ebp
0x08048c12 <next+3>:    sub    $0x28,%esp
0x08048c15 <next+6>:    mov    0x8(%ebp),%eax   <----|
0x08048c18 <next+9>:    mov    %eax,(%esp)           |
0x08048c1b <next+12>:   call   0x80486e9 <nextt><----|---- Verarbeitung des Wortes
0x08048c20 <next+17>:   mov    %eax,-0xc(%ebp)       |
0x08048c23 <next+20>:   cmpl   $0xc,-0xc(%ebp)       |<-- Schleife für jedes Wort
0x08048c27 <next+24>:   je     0x8048c15 <next+6>  --|
0x08048c29 <next+26>:   mov    -0xc(%ebp),%eax
0x08048c2c <next+29>:   leave
0x08048c2d <next+30>:   ret
```

Funktion nextt

```
0x0804894b <nextt+610>: je     0x804896a <nextt+641>
0x0804894d <nextt+612>: cmp    $0x6e,%eax
0x08048950 <nextt+615>: jg     0x804895e <nextt+629>
0x08048952 <nextt+617>: cmp    $0x22,%eax
0x08048955 <nextt+620>: je     0x80489ab <nextt+706>
0x08048957 <nextt+622>: cmp    $0x5c,%eax
0x0804895a <nextt+625>: je     0x80489ae <nextt+709>
0x0804895c <nextt+627>: jmp    0x804898e <nextt+677>
0x0804895e <nextt+629>: cmp    $0x72,%eax
0x08048961 <nextt+632>: je     0x8048982 <nextt+665>
0x08048963 <nextt+634>: cmp    $0x74,%eax
0x08048966 <nextt+637>: je     0x8048976 <nextt+653>
0x08048968 <nextt+639>: jmp    0x804898e <nextt+677>
```

In *nextt* finden wir die Hauptfunktionalität des Programms. Grundlegend haben wir hier ein riesiges Konstrukt aus *if-else* Anweisungen (siehe Listing). Innerhalb dieser Abfragen, befinden sich verschiedene Bereiche. Einer testet auf eine Zahl und gibt, falls der Test glückt 154 aus (Wir haben den reinen *isNumber* Test im Code nicht finden können, haben aber alle Integerwerte über ein Script getestet und sie führen alle zur selben Ausgabe). Im nächste wichtige Part führt mittels strcmp (@plt hinter dem Funktionsnamen bezieht sich darauf, dass diese Funktion relativ randomisiert im Speicher liegt und erst über eine Accesstable richtig aufgelst werden muss) ob der aktuelle Teil, der gelesen wird in unserem Array von möglichen Trefferwerten liegt. Der Vergleich muss über strcmp erfolgen, da bei strncmp die Trennung der Schlüsselwörter nicht duch Leerzeichen getrennt werden müsste. Ist es keiner, so wird nur das erste Zeichen genommen und ausgegeben. Ein normaler Buchstabe ergibt so immer die 408.

Funktion bad

Dieser Aufruf steht zu unterst in unserer Funktion *nextt* als allerletztes da. Sie wird ausgeführt, wenn alle anderen Tests fehlgeschlagen sind.

Da so gut wie jede Eingabe zu einer entsprechenden Ausgabe führt, wird diese Funktion so gut wie nie aufgerufen.

Die wenigen Fälle in denen das Programm abstürzt (so z.B. bei Eingabe '$'), wird die Funktion nicht aufgerufen (getestet mit gdb).

Ob die Funktion daher überhaupt angesprungen wird, konnten wir nicht bestätigen.

Funktion ung

Dieser Funktion geht es genau so wie *bad*. Wir haben keine Eingabe erreicht, die uns ermögliche diese Funktion anzuspringen, deshalb haben wir sie an dieser Stelle ignoriert.

Sonstige Funktionen

Ansonsten werden noch folgende Funktionen verwendet:

Get/Put Funktionen, die das ausgeben und lesen mit der Konsole erledigen. Soll man ein neues Wort (Wortfolge) eingaben, so wird immer vor *GET* angehalten.

getchar, ctype, printf, strcmp, strlen, strncmp : Aus den Standardfunktionen von C.