Log4c Writeup

Disclaimer

Bevor wir mit dem Writeup (Erklärung der Lösung) der Aufgabe beginnen, möchte ich anmerken, dass diese Aufgabe viele Dinge kombiniert und einiges an Vorwissen verlangt hat. Seid also nicht enttäuscht, falls ihr die Challenge nicht gelöst habt.

Bei jeglichen Fragen zu dem Writeup, zu Pwn / Binary Exploitation generell oder ähnlichem, schreibt mir **kampet / Niklas** einfach auf Discord :)

Beschreibung des Programms und Schwachstelle

Die Aufgabenbeschreibung hat schon vorgegeben, dass wir uns mit Name: “alice” und Passwort: “wonderland” einloggen können.

Sobald wir uns mit “alice” und “wonderland” eingeloggt haben, fragt uns das Programm nach dem Grund für unser Verbinden.

Unseren Input liest das Program mittels `fgets` ein und schreibt es ans Ende des `logMsg` Strings:

```
char logMsg[256] = "[%li] New login for user '%s', reason: ";
const int offset = 39;
if (!fgets(logMsg + offset, sizeof(logMsg) - offset, stdin))
```

Wenn wir nun `AAAAAAAAAA` eingeben, dann sieht die `logMsg` folgendermaßen aus:

```
"[%li] New login for user '%s', reason: AAAAAAAAAA;"
```

Diesen erzeugten String `logMsg` schreibt das Programm nun mittels der Funktion `fprintf` in eine Datei.

Die Schwachstelle liegt hierin, dass der von `fprintf` verwendete String vom User beeinflusst werden kann. Das ermöglicht sogenannte “Format String Exploits”.

Einführung printf

Die Funktion `fprintf`gehört zu der Klasse der `printf` Funktionen. Das besondere an diesen Funktionen ist es, dass sie die Vorkommnisse von z. B. `%s` im Text als Platzhalter interpretieren. Diese Platzhalter werden auch “format specifier” genannt, weil sie das Format definieren, wie die Dateien interpretiert werden können. Zum Beispiel gibt `%s` an, dass die Daten als String interpretiert werden sollen und `%d`, dass die Daten als Zahl interpretiert werden sollen.

printf Beispiel

Gegeben sei folgender Code:

```
printf("Erstes Argument: %s Zweites Argument: %s", "111", "222");

// Ausgabe: "Erstes Argument: 111 Zweites Argument 222"
```

Das erste `%s` wird hierbei durch “111” ersetzt und das zweite `%s` durch “222”.

Falls ich nicht auf diese Reihenfolge der format specifier angewiesen sein möchte, könnte ich auch folgendes schreiben:

```
printf("Zweites Argument: %2$s Erstes Argument: %1$s ", "111", "222");

// Ausgabe: "Zeites Argument: 222 Erstes Argument 111"
```

Ausnutzen der Schwachstelle

Der für diese Angriffe interessante format specifier ist `%n`. Wobei `%n` die Anzahl der bisher von `printf` ausgegebenen Zeichen an die Speicheradresse schreibt, die das Argument definiert.

Falls wir nun kontrollieren können, wie viele Bytes `fprintf` schreibt **und** die Adresse, an die geschrieben wird, kontrollieren können, dann können wir unseren Usernamen im Speicher umschreiben. Dadurch können wir unseren Namen von Alice zu Bob ändern und die Flagge erhalten.

Wenn wir uns den Code erneut durchlesen, sehen wir folgendes:

```
fprintf(logfile, logMsg, time(NULL), currentUser);
                                |          |____ zweites Argument
                                |_______________ erstes Argument
```

Das zweite Argument von `fprintf` enthält also unseren Usernamen! Wenn wir in unserem Input also `%2$n` angeben, dann sollten wir unseren Usernamen überschreiben.

Das können wir verifizieren, indem wir uns mit `alice` und `wonderland` einloggen und danach `%2$n` angeben:

```
username: alice
password: wonderland
your reason for connecting: %2$n
Hello, 1!
You are not an admin.
```

Anscheinend hat es funktioniert! Wir wurden nun mit `Hello, 1!` begrüßt, anstatt mit `Hello, alice!`.

Letzter Schritt zur Flagge

Wir wissen jetzt, dass der format specifier `%2$n` unseren Usernamen überschreibt, allerdings wird jetzt lediglich die Anzahl der bereits geschriebenen Bytes an die Speicheradresse geschrieben. Allerdings wird, unser Zielname `bob` im Speicher als ASCII kodierte Zeichen gespeichert. “b” wird zu 0x62 und “o” wird zu 0x6f. Insgesamt erhalten wir für den Namen “bob” die Zahl 0x626f62. (6451042 in Dezimal)

Aber wenn der format specifier `%2$n` nur die Anzahl der bislang geschriebenen Zeichen in die Speicheradresse schreibt und unsere `logMsg[256]` maximal 256 Bytes groß sein kann, wie sollen wir dann unseren Usernamen mit der Zahl 6451042 überschreiben? Das würde ja bedeuten, dass wir 6451042 Zeichen schreiben müssen?

Die Lösung für dieses Problem ist **Padding**. Und zwar lässt sich z. B. mit folgendem format specifier ein String mit Leerzeichen zur Länge 16 auffüllen:

```
printf("%16s", "Hallo Welt!");

// Ausgabe: "     Hallo Welt!" <- Von links auf 16 Zeichen aufgefüllt.
```

Somit können wir mit Leichtigkeit die Anzahl der geschriebenen Bytes auf 6451042 bringen!

Berechnung des Paddings

Aus unserem ersten Versuch mit dem Input `%2$n` wussten wir, dass unser Username mit dem Zeichen “1” überschrieben wurde, was der Zahl 49 entspricht. Wir berechnen die Differenz zu 645104:

```
6451042 - 49 = 6450993
```

Das bedeutet, dass unser Padding 6450993 Bytes groß sein muss.

Aus dem Code des Programms wissen wir, dass das erste Argument eine Zahl ist:

```
fprintf(logfile, logMsg, time(NULL), currentUser);
                                |          |____ zweites Argument
                                |_______________ erstes Argument
```

Diese Zahl können wir für unser Padding benutzen.

Erinnere, dass wir mit dem format specifier `%d` Integer darstellen können.

Unser endgültiger Payload besteht aus `%1$6450993d`, was das erste Argument auf eine Länge von 6450993 auffüllt. Und dem zweiten Teil `%2$n`, welcher die Anzahl der bisher geschriebenen Zeichen in unseren Usernamen reinschreibt. Wir erhalten zusammen den Payload:

```
%1$6450993d%2$n
```

Aufgrund des Paddings sollten wir insgesamt 6451042 Zeichen schreiben und demnach unseren Usernamen mit “bob” überschreiben:

```
Welcome to Bob's server. Please log in.
username: alice
password: wonderland
your reason for connecting: %1$6450993d%2$n
Hello, bob!
Here's the flag: SSH{w3_aRE_SaFe_fr0m_Log4j_We_onlY_u5e_printf-KS17s901IU7fyIjikFIYt}
```

Juhu! Es hat geklappt :)

Interessante Links

- Playlist von LiveOverflow zu Binary Exploitation / Pwn. Die Playlist ist etwas alt, allerdings immer noch empfehlenswert, um die Basics zu lernen. (anfängerfreundlich)
- Format String Exploits by pwn.college (Fortgeschritten)