Security Lab – Password-Cracking

VMware

Dieses Lab müssen Sie mit dem **Kali Linux-Image** durchführen.

1 Einleitung

In diesem Praktikum werden Sie verschiedene Methoden für das Offline Password-Cracking kennenlernen und anwenden, um Systempasswörter und die verwendeten Passwörter in einem Challenge-Response-basierten Authentisierungsprotokoll zu knacken.

2 Einführung in John the Ripper[1]

John the Ripper (oder kurz *john*) ist ein Open Source Password-Cracking Tool. Das Tool kann auf verschiedenen Plattformen verwendet werden und ist auf dem Kali Linux-Image bereits installiert.

john wird in einem Terminal wie folgt verwendet:

```
$ john [options] password_file
```

password_file ist die Datei, welche die Passwörter beinhaltet, die Sie knacken wollen. Dies können z.B. Passwort-Hashes sein. Die möglichen Optionen, mit denen *john* gestartet werden kann, werden weiter unten bei den Beispielen erläutert.

john unterstützt vier verschiedene Modes, um Passwörter zu knacken:

Single Crack Mode

Bei diesem Mode werden Kandidaten wie der Login Name oder der richtige Name des Benutzers (wenn vorhanden) als mögliche Passwortkandidaten getestet. Ebenfalls werden auf diese Kandidaten einige Mangling-Rules angewandt (z.B. Variieren der Gross-/Kleinschreibung oder Anhängen einer Ziffer), um weitere Varianten zu testen.

Wordlist Mode

Für diesen Mode müssen Sie eine Wörterliste (auch Dictionary genannt) spezifizieren, die verschiedene «likely passwords» enthält (also Passwörter, die häufig benutzt werden). Solche Wörterlisten finden Sie auch im Internet[2]. Optional kann zudem angegeben werden, dass *john* die in der Liste enthaltenen Passwörter nicht nur direkt verwenden soll, sondern auch Variationen davon (diese werden ebenfalls durch Mangling-Rules erzeugt).

Incremental Mode

Dieser Cracking-Mode entspricht einer Brute-Force Attacke. Es werden alle möglichen Zeichenkombinationen (Buchstaben, Ziffern, Sonderzeichen) durchprobiert. Dieser Mode dauert natürlich potentiell sehr lange. Wenn Sie allerdings eine kurze Passwortlänge oder einen kleinen Zeichensatz wählen, kann der Mode innerhalb nützlicher Frist enden.

External Mode

Diesen Mode werden Sie wahrscheinlich nicht verwenden. Der Vollständigkeit halber wird er hier aber trotzdem kurz erwähnt. Dieser Mode gibt Ihnen die Möglichkeit, einen eigenen Cracking-Mode zu definieren und diesen anzuwenden.

Beispiele

[1] https://www.openwall.com/john

[2] z.B. unter https://www.openwall.com/wordlists

Im Folgenden sind ein paar Beispiele für den Gebrauch von *john* aufgelistet:

```
$ john password_file
```

Damit wird versucht, die Passwörter in der Datei *password_file* zu cracken. *john* verwendet dazu zuerst den Single Crack Mode, dann eine Wordlist (Default ist */usr/share/john/password.lst*, eine recht kleine Wörterliste mit 3'000 – 4'000 Einträgen) mit Mangling-Rules und schliesslich den Incremental Mode.

```
$ john --single password_file
$ john –-single password_file1 password_file2
```

Dies startet *john* im Single Crack Mode. Wie Sie sehen ist es auch möglich, mehrere Passwort-Files gleichzeitig zu verwenden.

```
$ john --wordlist=password.lst --rules password_file
```

Dies startet *john* im Wordlist Mode mit Mangling-Rules. Als Wordlist können Sie dabei irgendeine Wordlist-Datei verwenden.

```
$ john –-incremental[=mode] password_file
```

Dies startet *john* im Incremental Mode. Es gibt mehrere verschiedene Incremental Modes, z.B.:

- *all* – Komplettes US-ASCII Character Set (95 Zeichen); es werden alle damit möglichen Passwörter mit Länge 1 bis 8 Zeichen durchprobiert.
- *alpha* – 26 Zeichen (Kleinbuchstaben); Passwortlängen von 1 bis 8 Zeichen.
- *digits* – 10 Zeichen (Ziffern von 0-9); Passwortlängen von 1 bis 8 Zeichen.
- *alnum* – Kombiniert die Modes *alpha* und *digits*.

Der Default-Mode ist dabei all. Um *john* in einem bestimmten Incremental Mode zu starten, z.B. *alpha*, führen Sie folgendes aus:

```
$ john –-incremental=alpha password_file
```

Ebenfalls kann ein Benutzer angegeben werden, wenn nur dessen Passwörter in *password_file* geknackt werden sollen:

```
$ john –-incremental –-users=user password_file
```

Dies startet *john* im Incremental Mode, es wird jedoch nur versucht, die Passwörter des Benutzers *user* zu cracken. Die Angabe von *--users* kann übrigens bei allen Modes verwendet werden.

Session wiederherstellen

john bietet die Möglichkeit, bestehende Cracking-Sessions zu unterbrechen, um sie zu einem späteren Zeitpunkt wieder fortzusetzen. Sessions können mit *Ctrl-C* unterbrochen werden. Sie können eine unterbrochene Session wie folgt fortsetzen:

```
$ john --restore password_file
```

Dabei ist *password_file* die Passwort-Datei, auf welcher Sie den Cracking-Vorgang fortsetzen wollen.

Geknackte Passwörter anschauen

john speichert geknackte Passwörter in der Datei *john.pot* ab (*~/.john/john.pot*). Um eine schöne Ausgabe aller bisher gecrackter Passwörter zu erhalten, geben Sie folgenden Befehl ein:

```
$ john --show password_file
```

password_file ist die Passwort-Datei, für welche die zugehörigen gecrackten Passwörter angezeigt werden sollen.

Status einer Session anschauen

Wenn Sie während einer laufenden Session die Enter-Taste drücken, zeigt *john* die Passwörter an, die gerade ausprobiert werden.

Konfigurationsdatei

Sämtliche Einstellungen (z.B. Mangling-Rules und die Modes für den Incremental Mode) sind in */etc/john/john.conf* konfiguriert und können angepasst werden.

3 Linux Password-Cracking mit *john*

Um erste Erfahrungen mit *john* zu sammeln werden Sie die Benutzerpasswörter des Kali Linux-Image knacken. Nehmen Sie an, ein Angreifer hat sich Zugang zum System verschafft und Zugriff auf die Passwort-Hashes erhalten und möchte diese nun offline knacken.

Die relevanten Dateien sind die folgenden:

- */etc/passwd* beinhaltet die Benutzernamen mit (wenn spezifiziert) zusätzlichen Informationen wie z.B. Vor- und Nachname.
- */etc/shadow* beinhaltet die Passwort-Hashes und die Salt-Werte. Je nach Linux-Version werden die Hashes unterschiedlich berechnet[3].

Als erstes müssen die beiden Dateien zusammengefügt werden. Führen Sie dies gleich aus. Dies geschieht mit dem untenstehenden Befehl (in einem Terminal). Da */etc/shadow* nur von *root* lesbar ist, müssen Sie zusätzlich den *sudo* Befehl verwenden (und das «root» Passwort eingeben, je nach umgebung):

```
$ sudo unshadow /etc/passwd /etc/shadow > linux_password_hashes
```

linux_password_hashes beinhaltet nun die zusammengefügten Dateien. Schauen Sie die erzeugte Datei kurz an, sie enthält insbesondere die Benutzernamen, die Salt-Werte, die Passwort-Hashes und Vor- und Nachname wenn vorhanden.

Wir müssen jetzt eine grössere Wordlist benutzen. Dekomprimieren Sie dazu /usr/share/wordlists/rockyou.txt.gz mit «gunzip» (als root). Anschliessend können Sie *john* starten:

```
$ john --rules=single --format=md5crypt
--wordlist=/usr/share/wordlists/rockyou.txt linux_password_hashes
```

Hinweis: Die sechs anzugreifenden Hashes sind noch alte MD5 Hashes. Die aktuellen Hashes der User «kali» und «root» kann man nur mit erheblich grösserem Aufwand angreifen.

Bereits nach kurzer Zeit wird *john* einige Passwörter gefunden haben. Beim Drücken der Enter-Taste meldet *john* jeweils, welche Passwörter gerade ausprobiert werden. Ebenfalls wird angegeben, wieviele % des aktuellen Modes bereits abgearbeitet wurden und wann der Mode voraussichtlich beendet sein wird (ETA). Wird ein Passwort gefunden, dann wird die ETA typischerweise reduziert, da ja nun ein Passwort-Hash weniger getestet werden muss. Ebenfalls wird ausgegeben, wieviele Passwörter pro Sekunde getestet werden (*c/s*). Lassen Sie *john* eine gewisse Zeit laufen (mind. fünf Passwörter sollten

[3] Aktuell wird eine Hashfunktion namens SHA-512 verwendet, um aus Passwort und Salt-Wert den Hash zu berechnen. Diese Hashfunktion liefert einen Hash von 512 bit Länge und wird ausserdem noch 5000 mal iteriert. Ist also p das Passwort, speichert Linux $H(H(H(\ldots H(\text{p})\ldots)))$ ab, wobei hier 5000 mal H = SHA-512 aufgerufen wird.

Sie finden, das kann 20 Minuten dauern), Sie können ja bereits mit den weiteren Aufgaben weitermachen.

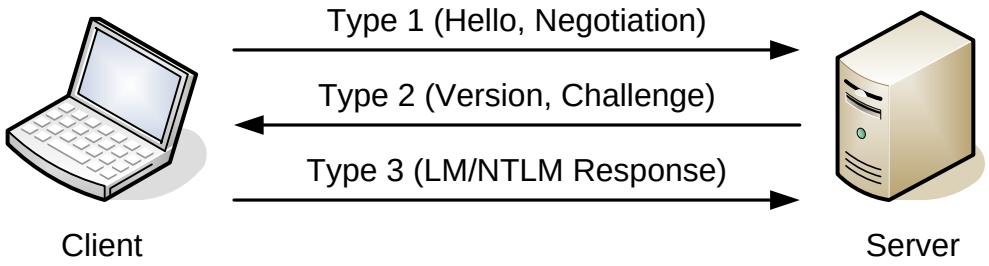
Insgesamt gibt es sechs Benutzer. Geben Sie in der folgenden Tabelle an, welche Passwörter *john* gefunden hat.

Username	**Passwort**
Andy	Hugentobler5
Eddie	Garfield
James	pinkfloyd9
Snoopy	015168
Superman	
Thomas	samantha

4 Authentisierung mit NTLM

Den Rest des Praktikums widmen Sie dem NTLM-Protokoll (NT LAN Manager) und wie man dabei verwendete Passwörter knacken kann. NTLM basiert auf dem früheren Protokoll LM (LAN Manager) und wurde von Microsoft entwickelt. NTLM war vor Windows 2000 das primäre Authentisierungsverfahren in Windows-Domänen. Seither übernimmt diese Rolle Kerberos, allerdings wird NTLM auch in den modernsten Windows-Versionen (und auch auf anderen Systemen) aus Gründen der Rückwärtskompatibilität nach wie vor unterstützt und auch oft eingesetzt. In Nicht-Domänen-Umgebungen wird für die Peer-to-Peer Authentisierung (direkter Zugriff eines Windows-Rechners auf einen anderen) immer NTLM verwendet. Der häufigste Anwendungsfall in diesem Zusammenhang ist das File/Directory Sharing, das auf den Protokollen SMB (Server Message Block) bzw. CIFS (Common Internet File System) basiert.

NTLM gibt's in verschiedenen Versionen und alle sind immer noch im Einsatz: die erste Version NTLMv1 und die zweite Version NTLMv2. Daneben gibt es noch eine Version mit dem Namen NTLM Session Security (manchmal auch als NTLMv2 Session Security bezeichnet). Der Ablauf der Authentisierung ist prinzipiell in allen Versionen (inklusive dem Vorgänger LM) derselbe. Das Protokoll ist ein Challenge-Response Verfahren und ist nachfolgend dargestellt:



Damit sich der Client beim Server authentisieren kann, registriert sich der Client zunächst beim Server mit einem gehashten Passwort.

Wenn ein Client auf einen Server zugreifen möchte (z.B. um auf einen File-Share zuzugreifen), so initiiert der Client den Authentisierungsvorgang gemäss obiger Abbildung. Dazu schickt er zuerst eine sogenannte Type 1 Nachricht an den Server. Diese sagt im wesentlichen «Hallo, ich möchte Deinen Dienst nutzen. Ich benutze NTML Version *V*».

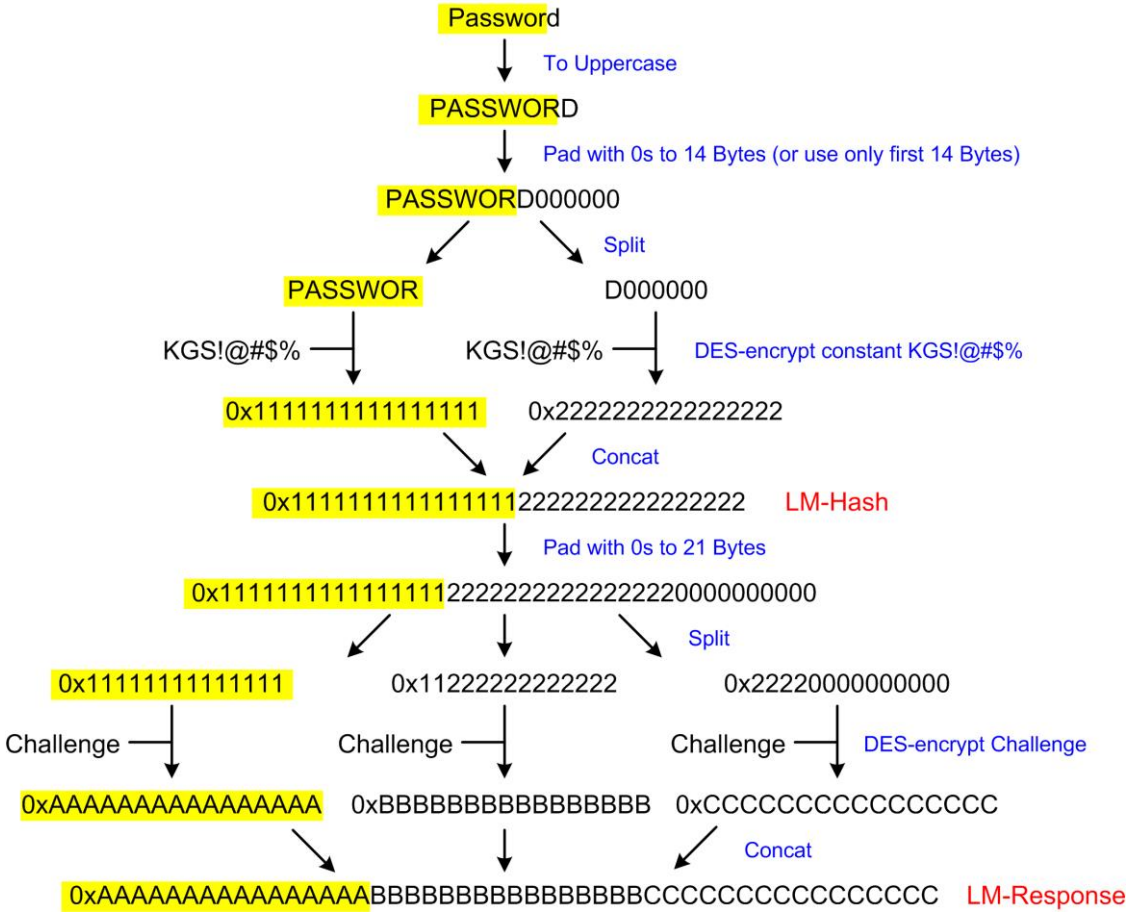
Ein Server antwortet auf die Anfrage mit einer Type 2 Meldung. Diese enthält neben der vom Server unterstützen Version *V'* auch eine *challenge*, in diesem Fall einen zufälligen Bitstring.

Um sich zu authentifizieren, verwendet der Client sein gehashtes Passwort als Schlüssel, um die Challenge zu verschlüsseln. Diese verschlüsselte Challenge schickt er als *Response* an den Server zurück. Sie haben noch keine Verschlüsselungsalgorithmen kennengelernt. Für dieses Praktikum ist es nur wichtig, dass Sie wissen, dass man ohne Kenntnis des Schlüssels (also des gehashten Passworts) die richtige Response zu einer gegebenen Challenge nicht berechnen kann.

Auf dieses Verfahren kann man nun eine Attacke durchführen, die im Prinzip mit jedem Passwort-basierten Challenge-Response Protokoll funktioniert. Man snifft den ganzen Authentisierungsvorgang und erhält dadurch die verwendete Version, die Challenge und die Response. Nun lassen sich in bekannter Manier Dictionary- und Brute-Force-Attacken darauf ausführen, indem ein Passwort gewählt wird, der resultierende LM- oder NT-Hash berechnet wird und die Response basierend auf der Challenge berechnet wird. Entspricht die berechnete Response der gesnifften Response, so hat man das Passwort gefunden. Falls man als Angreifer die Challenge im Voraus kennt oder diesen selbst wählen kann, kann man zudem für alle möglichen Passwörter (zumindest bis zu einer gewissen Passwortkomplexität, je nach verfügbarer Rechenpower) die resultierenden Responses vorausberechnen und sogenannte Rainbow Tables verwenden. Im Allgemeinen gilt natürlich, dass die Attacken umso aufwändiger werden, je komplexer das herauszufindende Passwort ist.

Berechnung der LM-Response

LM weist eine grundlegende Schwachstelle auf, die wir später ausnutzen wollen. Sie liegt einerseits in der Berechnung des LM-Hash aus dem Passwort und andererseits der Berechnung der LM-Response aus der Challenge und dem LM-Hash. Der Ablauf ist in der nachfolgenden Abbildung dargestellt und zeigt, wie aus einem Passwort *Password* zuerst der LM-Hash und dann damit basierend auf der Challenge des Servers die LM-Response berechnet wird. (Lassen Sie sich nicht davon irritieren, dass hier von «DES-Verschlüsselung» gesprochen wird. Sie müssen hier nichts über Verschlüsselung wissen. Wenn Sie das aber umtreibt, behandeln Sie hier «DES-Verschlüsselung» einfach wie einen weiteren Kasten, ähnlich wie eine Hashfunktion: Gleiche Eingaben geben gleiche Ausgaben, verschiedene Eingaben geben zufällige Ausgaben.)



Dabei sind insbesondere zwei Dinge auffällig: Zum einen wird für die Berechnung des LM-Hash das Passwort in Grossbuchstaben umgewandelt, wodurch jedes Passwort signifikant an Stärke verliert (Passwörter dürfen auch Ziffern und Sonderzeichen erhalten, die von dieser Umwandlung in Grossbuchstaben nicht betroffen sind). Zum anderen führen die ersten sieben Zeichen des Passworts direkt zum ersten Drittel der LM-Response ohne dass dies von den weiteren Zeichen des Passworts beeinflusst wird; siehe dazu die gelb unterlegten Zeichen in der obigen Abbildung. Dies heisst, dass diese sieben Zeichen später komplett separat geknackt werden können. Da ein Passwort höchstens 14 Zeichen beinhalten kann (allfällig weitere Zeichen werden beim Bilden des LM-Hash einfach ignoriert), muss im zweiten Schritt dann nochmals ein Passwort mit sieben Zeichen geknackt werden. Das unabhängige Knacken von zwei Passwörtern mit sieben Zeichen ist natürlich sehr viel einfacher als alle 14 Zeichen gemeinsam zu knacken.

Berechnung der NTLMv1-Response

NTLMv1 macht die Sache nicht viel besser, aber immerhin werden im ersten Schritt die Kleinbuchstaben im Passwort *nicht* in Grossbuchstaben umgewandelt. Zuerst wird ein MD4-Hash über dem Passwort berechnet, was in einem 128-bit langen Wert resultiert. Diese 16 Bytes werden mit 5 Null-Bytes auf 21 Bytes verlängert, woraus drei Gruppen mit je 7 Bytes generiert werden. Diese drei 7-byte Werte werden dann wie bei LM oben dargestellt verwendet, um die Challenge dreimal mit DES zu verschlüsseln.

Ein Angreifer, der Challenge und zugehörige Response abfängt, muss daher «nur» zwei DES-Verschlüsselungen knacken, um die ersten 14 Bytes des MD4-Hashes zu erhalten. Das Knacken der dritten DES-Verschlüsselung ist trivial, weil alle Bytes ausser den ersten beiden Null-Bytes sind. Der Angreifer erhält damit zwar nur den MD4-Hash und nicht das dahinterliegende Passwort, aber der MD4-Hash reicht dem Angreifer, um sich im Namen des Benutzers mit NTLMv1 zu authentisieren[4]. Das Knacken eines DES-Schlüssels ist natürlich nach wie vor nicht ganz trivial, in Abschnitt 7 werden Sie aber noch erfahren, dass dies auch für Privatpersonen gegen Bezahlung problemlos machbar ist.

Berechnung der NTLMv2-Response

NTLMv2 (und auch NTLM Session Security) weist die Schwachstellen von LM und NTLMv1 nicht auf. Ein weiterer Vorteil ist, dass auch der Client eine Challenge für die Berechnung der Response beisteuert (es ist bei Sicherheitsprotokollen immer eine gute Idee, wenn beide Parteien Zufallsdaten beisteuern, damit z.B. ein Angreifer, der sich als Server ausgibt, nicht einfach alle beigesteuerten Zufallsdaten selbst wählen kann). Ein schlecht gewähltes Passwort ist aber natürlich trotzdem einfach knackbar.

In den nächsten Abschnitten werden Sie nun verschiedene Passwort-Cracking Methoden auf NTLM anwenden.

5 NTLM Attacke Teil 1: Dictionary Attacke auf NTLMv2

Es gibt mehrere Möglichkeiten, wie ein Angreifer an die NTLM-Challenges und -Responses gelangen kann, z.B.:

- Als Man-in-the-Middle (MITM) die Kommunikation zwischen Client und Server sniffen, wodurch die komplette Authentisierung und insbesondere die Challenges und zugehörigen Responses erhalten werden. MITM kann man – wie Sie in einem anderen Praktikum gelernt haben oder noch lernen werden – sehr einfach mittels ARP Spoofing (z.B. mit dem Tool Ettercap) werden. Leider gibt Ettercap die relevanten Authentisierungsdaten (Challenges & Responses) nicht aus, wodurch man diese selbst aus den aufgezeichneten Daten extrahieren muss.

[4] https://www.microsoft.com/security/blog/2012/12/11/new-guidance-to-mitigate-determined-adversaries-favorite-attack-pass-the-hash/

- Man gibt sich als Server aus und bringt einen Benutzer dazu, einen Anmeldevorgang durchzuführen. Dazu könnte man beispielsweise dem Benutzer eine E-Mail senden mit einem \\SERVER\SHARE Link. Je nach E-Mail Client wird automatisch oder nach einem Klick des Benutzers auf den Link eine Verbindung aufgebaut und damit eine Authentisierung durchgeführt.

Weil Ettercap wie gesagt die erste Variante nicht komfortabel unterstützt und damit Sie noch ein weiteres Tool kennenlernen, verwenden wir hier die zweite Variante.

Für den Server verwenden wir das *Metasploit Framework*[5]. Metasploit ist ein sehr mächtiges Framework zum Ausnutzen diverser Schwachstellen. In unserem Fall wird Metasploit einen Samba-Server simulieren, bei welchem sich Benutzer mit NTLM authentisieren sollen.

Starten Sie in einem Terminal die Metasploit Framework Console – *msfconsole* (das dauert ein bisschen), die *root*-Rechte benötigt:

```
$ sudo msfconsole
```

Simulieren Sie dann einen Samba-Server, um Responses zu sammeln, indem Sie innerhalb von *msfconsole* folgendes eingeben:

```
use auxiliary/server/capture/smb
```

Als nächstes spezifizieren Sie, dass die verwendeten Challenges & Responses in Dateien mit dem Präfix *john* auf dem Desktop abgespeichert werden:

```
set JOHNPWFILE /home/kali/Desktop/john
```

Diese Dateien können dann von *john* verwendet werden, um die Passwörter zu knacken.

Mit

```
info
```

können Sie die aktuellen Settings anzeigen.

Mit

```
run
```

wird der Server gestartet und mit

```
netstat -ntlp
```

können Sie prüfen, dass der Server wirklich auf TCP Port 445 horcht.

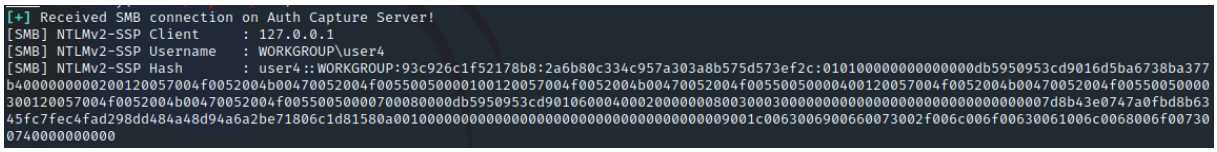
Benutzerseitig verwenden wir *smbclient* als Samba-Client. Melden Sie sich (am besten in einem anderen Terminal-Fenster) als *user4* mit Password *springfield* beim Server an:

```
$ smbclient //localhost/share -U user4
```

Die Anmeldung wird fehlschlagen, da es dem Metasploit Samba-Server ja nur darum geht, die Responses zu erhalten. Zudem existiert Benutzer *user4* (und auch die im Folgenden verwendeten Benutzer) auf dem System gar nicht.

Die *msfconsole* sollte einen Output der folgenden Art liefern:

[5] https://www.metasploit.com



Was erkennt man hier:

- Es wird NTLMv2 verwendet – also die sichere NTLM-Variante. *smbclient* ist also standardmässig so konfiguriert, dass NTLMv2 verwendet wird.

Auf dem Desktop sollte zudem ein File aufgetaucht sein, *john_netntlmv2*. Dieses beinhaltet den Benutzernamen, die verwendeten Challenges und die zugehörige Response in einem Format, so dass die Files von *john* verarbeitet werden können:

```
user4::WORKGROUP:1122334455667788:958fa99bc0f7eae891a2612e38f79
0cc:0101000000000000f9d159cb11cf01fd0bb74bb300e30c00000000020
0120057004f0052004b00470052004f00550050000000000
```

Am Anfang (nach *WORKGROUP:*) steht dabei die Challenge des Servers.

Verwenden Sie nun *john*, um das Passwort zu knacken. Wechseln Sie dazu im Terminal am besten in das *Desktop* Verzeichnis (*/home/kali/Desktop*), damit Sie direkt auf die von *msfconsole* erzeugten Files zugreifen können. Als Wordlist verwenden Sie eine längere Liste als die Default-Wordlist, sie enthält rund 88'000 Einträge:

```
/usr/share/wordlists/metasploit/password.lst
```

Wie lautet der Aufruf von *john* (verwenden Sie nur den Wordlist Mode, siehe Abschnitt 2) und wie lange dauert es, bis *john* das Passwort gefunden hat? Hinweis: Sie müssen zuvor die evtl. noch laufende *john* Cracking-Session aus Abschnitt 3 beenden.

sudo john --rules=single --wordlist=/usr/share/wordlists/metasploit/password.lst john_netntlmv2

Es hat nur eine Sekunde gedauert!

john kann also auch NTLMv2-Responses knacken und Sie sehen auch, dass die Attacke in diesem Fall sehr effizient ist. Für jedes Passwort aus der Wörterliste hat *john* basierend auf den Challenges von Client und Server die NTLMv2-Response berechnet und mit dem erwarteten Wert verglichen. Stimmen die Werte überein, ist das Passwort gefunden. Man beachte auch, dass die Rate der getesteten Passwörter nun viel höher als bei den Linux Systempasswörtern ist und sie sollte «einige 1’000’000» betragen. Dies hat damit zu tun, dass nun pro getestet Passwort nur ein MD4-Hash und eine HMAC-MD5 Operation durchgeführt werden muss, was sehr effizient gemacht werden kann.

Machen Sie einen weiteren Versuch mit *user5* und *springfield5* als Passwort. Wie lautet der Aufruf von *john* und wie lange dauert es, bis *john* das Passwort gefunden hat? Wenn das Passwort nicht gefunden wird, dann haben Sie vermutlich eine Option vergessen, siehe Abschnitt 2.

3 Sekunden

Auch dieses Passwort wurde also sehr schnell gefunden. Verwenden Sie nun *user6* und *springfield55* und versuchen Sie erneut, dass Passwort gleich wie oben zu knacken. Was beobachten Sie? Wieso könnte dies so sein?

Es geht länger! 5min oder mehr

Diese Limitierung können Sie umgehen, indem Sie die Mangling-Rules von *john* anpassen. Diese Rules verwenden eine sehr umfangreiche Syntax, Details finden Sie unter

```
https://www.openwall.com/john/doc/RULES.shtml
```

Die Regeln sind im Konfigurationsfile

```
/etc/john/john.conf
```

definiert und können beliebig adaptiert werden. Öffnen Sie die Datei (mit *sudo*) und suchen Sie die Wordlist-Rules (*[List.Rules:Wordlist]*). Ein paar Zeilen weiter finden Sie folgende Regel:

```
# Lowercase pure alphabetic words and append a digit or simple punctuation
<* >2 !?A l $[2!37954860.?]
```

Dies ist offensichtlich die Regel, mit welcher *springfield5* gefunden wurde. Die Regel versteht sich wie folgt:

- `<* >2`: Die Regel gilt für Wörter mit mehr als 2 Zeichen (`>2`); nach oben gibt es keine Grenze (`<*`).
- `!?A`: Die Regel soll nicht verwendet werden, wenn das Wort ein Zeichen der Klasse `?A` enthält. `?a` bezeichnet die Klasse aller Buchstaben (a-z, A-Z) und `?A` negiert diese Klasse (alle Zeichen ausser Buchstaben). Enthält das Wort also irgendwelche Zeichen, die nicht Buchstaben sind, so wird die Regel nicht verwendet.
- `l`: Wandelt das Wort in Kleinbuchstaben.
- `$[2!37954860.?]`: Hängt ein Zeichen aus der Liste an. Hier wird eine Regex-ähnliche Syntax verwendet, die eckigen Klammern bedeuten dabei «eines aus der Liste».

Fügen Sie nun eine Regel hinzu, die das Wort in Kleinbuchstaben wandelt und zwei Ziffern anhängt. Dies ist basierend auf der obigen Regel einfach:

```
# Lowercase pure alphabetic words and append two digits
<* >2 !?A l $[0-9] $[0-9]
```

Versuchen Sie nun erneut, das Passwort zu knacken. Was ist das Ergebnis?

Es ging viel schneller. (1 min)

Beurteilen Sie kurz den Effekt dieser Änderung der Regeln. Wenn die Wörterliste 100'000 Einträge hat, wie viele zusätzliche Passworttests haben Sie mit dieser Konfigurationsänderung eingeführt? Was sollten Sie deshalb bei jeder Regel, die Sie hinzufügen, beachten?

Nicht unnötig viel Zeit vergeuden für zu viele Tests!

100'000 * 10! Weil 10 verschiedene Zahlen am Schluss

6 NTLM Attacke Teil 2: Brute-Force Attacke auf NTLMv2

Starten Sie eine weitere Authentisierung, diesmal mit *user7* und Passwort *kyxb*. Dieses Wort ist kaum in einer Wörterliste, also ist hier ein Brute-Force Attacke wohl erfolgreicher – dazu bietet *john* ja den Incremental Mode. Schauen wir zuerst wieder das Konfigurationsfile an.

Suchen Sie den Bereich mit den Incremental Modes *(# Incremental modes*). Etwas weiter unten finden Sie den Mode *[Incremental:Lower]*, damit werden alle möglichen Kombinationen von 1-13 Kleinbuchstaben getestet – was vermutlich zu lange dauern würde, um in nützlicher Zeit ans Ziel zu gelangen (*lower.chr* bezeichnet den Zeichensatz bestehend aus Kleinbuchstaben).

Fügen Sie deshalb einen eigenen Mode hinzu, um Passwörter mit *1-4 Kleinbuchstaben* zu testen. Die Konfiguration sieht wie folgt aus, sie sollte selbsterklärend sein:

```
[Incremental:Lower15]
File = $JOHN/lower.chr
MinLen = 1
MaxLen = 4
CharCount = 26
```

Basierend auf dem *c/s* Wert auf Ihrem System, wie lange wird es ungefähr dauern, alle Passwörter zu testen?

552'573 c/s

Max. Kombinationen = 26^1 + 26^2 + 26 ^3 + 26 ^4 = 475'254
Anzahl Sekunden = 1s*474'254/552'573

Führen Sie die Attacke durch. Verwenden Sie bei *john* die Option

```
--incremental=Lower15
```

Konnten Sie das Passwort finden? Stimmt die Suchdauer mit der obigen Abschätzung überein?

Ja, weniger als eine Sekunde

john --incremental=Lower15 --users=user7 john_netntlmv2

Längere Passwörter oder das Verwenden eines grösseren Charsets wird die Brute-Force Attacke schnell ans Limit bringen. Betrachten wir als nächstes ein Passwort, dass statt nur 4 Kleinbuchstaben ein 4-stelliges Passwort mit Klein- und Grossbuchstaben und Ziffern verwenden. Verwenden Sie dazu

user8 und Passwort *v7Zm*. Wie viele mögliche Passwörter gibt es? Wie lange wird es in etwa dauern, alle Varianten durchzuprobieren? Betrachten Sie nur Passwörter der Länge 4 Zeichen:

Weniger als eine Sekunde, 2^24

Auch hier brauchen Sie einen Mode, der die entsprechenden Zeichen durchtestet. *john* bietet einen Mode *[Incremental:Alnum]*, die das Charset *alnum.chr* bestehend aus Kleinbuchstaben, Grossbuchstaben und Ziffern verwendet. Wie oben werden auch hier alle Kombinationen von 1-13 Zeichen getestet, was wiederum kaum in nützlicher Zeit zum Ziel führen würde. Auch hier kann man aber einen besser passenden Mode definieren, der Passwörter mit genau 4 Zeichen testet:

```
[Incremental:Alnum4]
File = $JOHN/alnum.chr
MinLen = 4
MaxLen = 4
CharCount = 62
```

Wir machen uns es hier aber noch etwas «komplizierter», damit Sie ein weiteres Feature des Incremental Mode kennenlernen. Nehmen Sie an, es gibt für den Fall kein passendes Charset, aber immerhin gibt es eines, das ein Subset der gewünschten Zeichen enthält. In diesem Fall können Sie mit der Option *Extra* weitere Zeichen hinzufügen.

Wir verwenden dies, um das Charset *lowernum.chr*, das Kleinbuchstaben und Ziffern beinhaltet, mit Grossbuchstaben zu ergänzen:

```
[Incremental:Alnum4]
File = $JOHN/lowernum.chr
Extra = ABCDEFGHIJKLMNOPQRSTUVWXYZ
MinLen = 4
MaxLen = 4
CharCount = 62
```

Fügen Sie diesen Mode hinzu und knacken Sie das Passwort. Lassen Sie *john* rechnen und fahren Sie mit dem Praktikum weiter. Wie lange hat es effektiv gedauert? Stimmt das mit Ihrer Erwartung überein?

Eine Sekunde

Das ist natürlich immer noch überhaupt nicht sicher, aber immerhin etwas besser als die 4 Kleinbuchstaben. Jedes zusätzliche Zeichen erhöht die Passwortstärke enorm: Bei 6 Zeichen (Klein- und Grossbuchstaben und Ziffern) sind wir bei 57 Milliarden Kombinationen, was auf einem State-of-the-Art PC etwa 6 Stunden benötigt. 7 Zeichen entsprechen 3.5 Billionen Kombinationen oder rund 15 Tage und ab 8 Zeichen nähern wir uns so langsam dem Bereich, wo der benötigte Rechenaufwand für «Privatpersonen» doch langsam sehr aufwändig wird (220 Billionen Kombinationen und ca. 2.5 Jahre). Denken Sie aber auch daran, dass man die Brute-Force Attacke (wie jede «Passwort-Ausprobier-Attacke») perfekt auf mehrere Rechner verteilen kann und gewisse Institutionen (NSA etc.) vermutlich über

enorme Rechenpower verfügen (bei 1'000 PCs dauert es dann nur noch rund einen Tag statt 2.5 Jahre). Zudem kann man günstige Rechenpower in der Cloud nutzen (siehe Abschnitt 7) oder auch spezielle Hardware anschaffen, um die Testrate massiv zu erhöhen. Für wirklich sensitive und wertvolle Daten, die einem Angreifer den Einsatz von «einigen PC-Jahren» Wert sind, sind 8 zufällig gewählte Zeichen also definitiv immer noch nicht sicher genug.

Als Fazit dieser Dictionary und Brute-Force Attacken können Sie folgendes mitnehmen: Auch wenn das verwendete Protokoll als sicher betrachtet wird, was bei dem hier verwendeten NTLMv2 grundsätzlich der Fall ist, so kann die Sicherheit durch schlecht gewählte Passwörter völlig kompromittiert weden.

7 Password-Cracking – Schlussbemerkungen

Sie haben gesehen, dass es heute mächtige Tools für das Offline Password-Cracking gibt. Sind Passwörter schlecht gewählt (Wörter aus einem Dictionary, allenfalls leicht verändert), dann ist es oft sehr einfach, diese in kurzer Zeit zu knacken. Auch kurze zufällige Passwörter sind schnell herausgefunden, auch wenn hier der Aufwand mit zunehmender Passwortlänge exponentiell ansteigt.

Die NTLM Downgrading Attacke sollte heute in den meisten Fällen nicht mehr möglich sein, da nur noch wenige Systeme im Einsatz sind, die LM per Default unterstützen (unter anderem Windows XP). Sie zeigt aber ein wichtiges Grundproblem auf, wenn es von Protokollen ältere, verwundbare Versionen gibt: Es dauert auch beim Erscheinen der neuen Version dann oft sehr lange (viele Jahre), bis die älteren Versionen dann wirklich weg sind und es gibt meist eine lange Übergangsphase, in der die alte und neue Version unterstützt wird – damit neuere und ältere Geräte miteinander verwendet werden können. NTLM ist hier nur ein Beispiel. Ein anderes bekanntes Beispiel ist SSL/TLS, wo man bei der Version SSLv2 gravierende Mängel festgestellt hat (Mitte der 1990-er Jahre) und mit SSLv3 schnell Abhilfe geliefert hat (in der Zwischenzeit gilt allerdings auch SSLv3 als gebrochen), aber auch heute noch finden sich vereinzelte Clients und Server, die nach wie vor SSLv2 unterstützen.

Praktikumspunkte

In diesem Praktikum können Sie **2 Praktikumspunkte** erreichen. Zeigen Sie dazu ihre Lösungen aus den vorherigen Aufgaben ihrem Betreuer.

Anhang 1: Einführung in Passwort-Hashing

Wir starten mit dem Modell der Hashfunktionen:

Stellen Sie sich eine Kiste vor, in der zwei Löcher sind. In das eine Loch können Sie *beliebig lange Bitstrings irgendeiner Art hineintun*, die man *Urbilder* (engl. *preimage*) nennt, und aus dem anderen Loch kommt *ein Bitstring fester Länge* (z.B. 256 bit) *heraus*, den man *Hash* nennt. Wenn Sie *dieselben Bitstrings hineintun, kommt derselbe Bitstring heraus*. Wenn Sie *verschiedene Bitstrings hineintun*, kommen Bitstrings heraus, die *weder untereinander noch mit den Eingaben irgendetwas zu tun haben*. Echte kryptographische Hashfunktionen verhalten sich wie diese Kiste. Beachten Sie aber, dass es sich hier nur um ein *Modell* handelt, also um eine Art, wie man über Hashfunktionen nachdenken kann. Realisiert werden solche Funktionen anders. Ist x ein preimage und h der zugehörige Hash der Hashfunktion H, dann schreibt man $h = H(x)$.

Kryptographische Hashfunktionen haben ein paar Eigenschaften. Diese hier brauchen wir für das Praktikum:

1. Aus einem Hash kann man kein preimage leicht berechnen. Ist also $h = H(x)$ gegeben, ohne dass man x kennt, ist man aufgeschmissen. Man nennt das *preimage-Resistenz.*
2. Natürlich lässt sich zu einem gegebenen Hash h ein Urbild durch Ausprobieren herstellen: man versucht so lange verschiedene x, bis man eines findet, für das $h = H(x)$ gilt. Sind alle möglichen Urbilder gleichwahrscheinlich, gilt: Bei einer Hashfunktion, die Hashes von n bit Länge produziert, muss man im Mittel 2^n Hashes berechnen, um ein Urbild auf diese Art zu finden. Wenn n gross genug ist, stehen Punkt 1 und 2 also nicht in Widerspruch.

Was hat das nun mit der Speicherung von Passwörtern zu tun? Wenn ich mich bei einem System anmelden möchte, das Passwörter für die Authentifikation nutzt, übermittle ich bei der Anmeldung meinen Benutzernamen und mein Passwort. Das System prüft dann, ob es einen Benutzer mit dieser Benutzername/Passwort-Kombination gibt. Ist das der Fall, wird die Person angemeldet, ansonsten nicht.

Um die Prüfung vornehmen zu können, muss das System irgendwie entscheiden können, ob das von mir übermittelte Passwort richtig ist. Eine einfache Möglichkeit besteht darin, eine Datei auf dem System zu führen, in der alle gültigen Benutzername/Passwort-Kombinationen verzeichnet sind. Gelangt aber ein Angreifer in den Besitz dieser Datei, besitzt er die Passwörter sämtlicher Accounts. Das möchte man also vermeiden

Eine Möglichkeit, dem zu begegnen, ist es, die Passwörter zu *verschlüsseln*. Das Problem dabei ist aber, dass dann der Schlüssel dem Programm vorliegen muss. Ein Angreifer, der in den Besitz des Schlüssels gelangt, könnte dann die Passwörter entschlüsseln und man wäre so weit wie vorher.

Eine andere Möglichkeit besteht darin, in dieser Datei die Passwörter zu *hashen*. Statt des Passworts p wird also dessen Hash $H(p)$ gespeichert. Melde ich mich nun mit meinem Passwort p an, dann berechnet das System $H(p)$ und schaut, ob Benutzername und $H(p)$ in der Datei zusammen vorkommt.

Der Vorteil dieses Verfahrens liegt darin, dass es hier nichts zu entschlüsseln gibt. Da die verwendete Hashfunktion preimage resistant ist, kann man aus den gespeicherten Hashes die Passwörter nicht zurückberechnen. Produziert die Hashfunktion genügend lange Hashes, sollte auch Ausprobieren verschiedener Passwörter nichts nutzen: bei n-bit Hashes braucht es im Mittel 2^n Hashversuche, um ein preimage zu finden (siehe Punkt 2 oben). Damit könnte man die Passwortdatei auch veröffentlichen. (Frühe Versionen von Unix haben genau das gemacht.)

Das Problem besteht jedoch darin, dass die obige Argumentation nur funktioniert, *wenn alle möglichen Passwörter gleich wahrscheinlich sind*. Da aber Leute eher ein Passwort wie „passwort“ oder „123456“ wählen statt „7/jJKLLm;1“, ist das offensichtlich nicht der Fall. Gelangt man also in den Besitz eines gehashten Passworts h, kann man nach folgendem Plan vorgehen, um das preimage, also das Passwort, herauszufinden:

1. Lade von irgendwo eine Liste mit oft benutzen Passwörtern herunter.
2. Sortiere die Liste absteigend in der Reihenfolge ihrer Popularität. Das am häufigsten verwendete Passwort ist also das erste, das am wenigsten häufige das letzte Passwort in der Liste.
3. Führe Schritte 4 und 5 für alle Passwörter p nach der Beliebtheitsreihenfolge aus:

4. Berechne h’ = H(p).
5. Ist h = h’, gib p aus und beende den Lauf. Das Passwort ist geknackt. Ansonsten nimm das nächste Passwort p von der Liste und kehre zu Schritt 4 zurück.

Wenn das zu knackende Passwort ein häufig benutztes Passwort ist, führt dieser Ansatz weitaus schneller zum Ziel, als man das eigentlich erwarten sollte, wenn Passwörter alle gleich wahrscheinlich wären. Natürlich kann man diesen Algorithmus noch verfeinern, indem man das Passwort, das auf der Liste steht, noch leicht verändert, etwa durch Gross- und Kleinschreibung, oder durch das Anhängen von ein paar Ziffern. Damit wären dann auch Passwörter wie etwa *SupErMan99* abgedeckt.

Im Rest des Praktikums probieren Sie das nun aus.

Was Sie noch wissen müssen: Aus Gründen, die wir erst in der Vorlesung *Data Integrity and Authentication* besprechen können, wird in der Passwortdatei nicht einfach der zu einem Passwort *p* gehörende Hash gespeichert. Stattdessen wird für jeden Benutzer ein anderer, zufälliger, Wert *s* bestimmt, der *salt*. Beim Hashen werden dann *s* und *p* aneinandergehängt. Gehasht wird also nicht *p*, sondern *s* + *p*. Gespeichert werden dann der Benutzername, der Salt *s* und der Hash $h = H(s + p)$. Bei der Anmeldung wird das Passwort *p* vom Benutzer übermittelt. Das System schaut dann den Salt *s* und den Hash *h* in der Passwortdatei nach, berechnet $H(s + p)$ und schaut, ob das mit dem gespeicherten *h* übereinstimmt. Wenn ja, wird der Benutzer angemeldet, sonst nicht. Der Begriff *Salt* taucht an einigen Stellen im Praktikum auf, wird aber zur Beantwortung der Fragen nicht benötigt.