

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is light green. They are positioned diagonally, with the blue one partially covering the green one.

Prozessinjektion

Hein, Happe, Katibeh



Ziel und Inhalt

Ziel der PG:

- Bau eines Debuggers mit Fokus auf gut lesbarem und dokumentierten Code
- Windows Funktion Create Remote Thread auf Linux 64 Bit portieren (Jugaad)

Wie ?

- Debugger ist in C für Linux 64 Bit geschrieben
- wichtigstes Werkzeug ist C-Funktion ptrace



Benötigtes Vorwissen

- Systemaufrufe in Linux

- Linker und Loader

- PIC

- GOT und PLT

- ASLR

- Signale



Systemaufrufe in Linux

- bieten Schnittstelle zwischen User-Mode und Kernel-Mode
- Werte werden in Register geladen
- RAX-Register wird mit Syscall Nummer belegt und ein Syscall wird ausgeführt
- Programmfluss wird an Kernel-Mode übergeben
- Registerwerte werden nach Linux Calling Convention ausgewertet
- Rückgabewert wird in Register geladen
- Programmfluss wieder an User-Mode

Systemaufrufe in Linux (2)

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count
2	sys_open	const char *filename	int flags	int mode
60	sys_exit	int error_code		

http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Integer value	Name	<unistd.h> symbolic constant ^[1]	<stdio.h> file stream ^[2]
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

https://en.wikipedia.org/wiki/File_descriptor

Systemaufrufe in Linux (3)

```
1  section .data
2
3      str: db "Hello World", 10
4      str_len equ $ - str
5      ; current adress - str label adress = length of "Hello World\n"
6
7  section .text
8  global _start
9
10 _start:
11
12      ; sys_write
13      mov RAX, 1
14      mov RDI, 1          ; stdout
15      mov RSI, str
16      mov RDX, str_len
17      syscall
18
19      ; sys_exit
20      mov RAX, 60
21      mov RDI, 0          ; error code
22      syscall
```



Linker

- verbindet abstrakte Namen mit konkreten Adresswerten (berechnet Offset)

Beispiele:

- Variable X wird der virtuelle Adresswert, wo X liegt, zugewiesen

- Funktion sqrt wird mit sqrt aus der math library assoziiert

- fügt alle Codeabschnitte zusammen



Loader

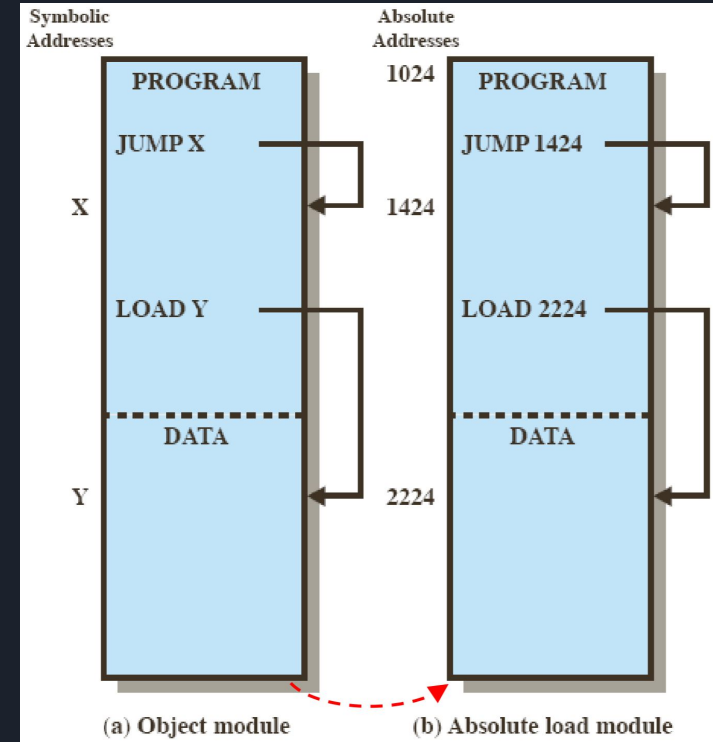
- lädt Code in den Arbeitsspeicher
- wählt Startadressen, sodass Code sich nicht überlappt

3 Loader Arten:

- absolute loading
- relocatable loading
- dynamic runtime loading

Absolute Loading

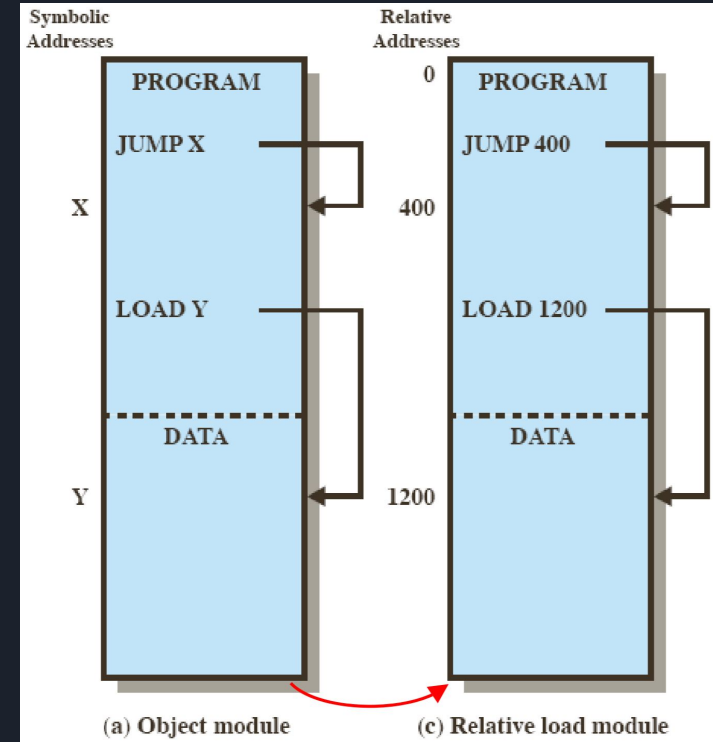
- lädt nur absolute Adressen (z.B bei JUMP X)
- bindet während Kompilierzeit
- setzt voraus, dass Code unverändert an der selben Stelle im Speicher liegt



Stallings, W. (2004) Operating Systems: Internals and Design Principles (5th Edition).

Relocatable Loading

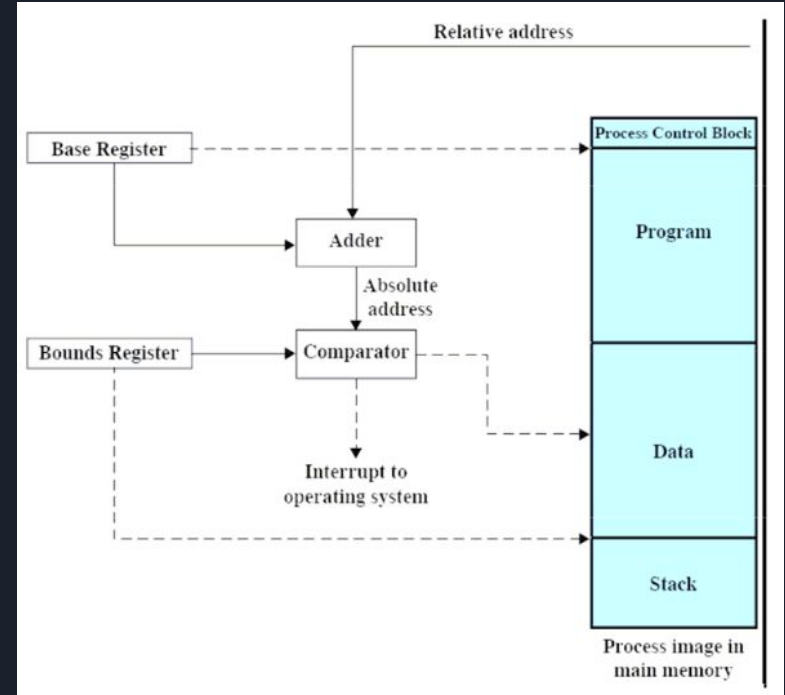
- lädt Adressen relativ zum Code
- bindet während Ladezeit
- setzt voraus, dass Code intern unverändert bleibt, kann aber an jeder Stelle des Speichers geladen werden



Stallings, W. (2004) Operating Systems: Internals and Design Principles (5th Edition).

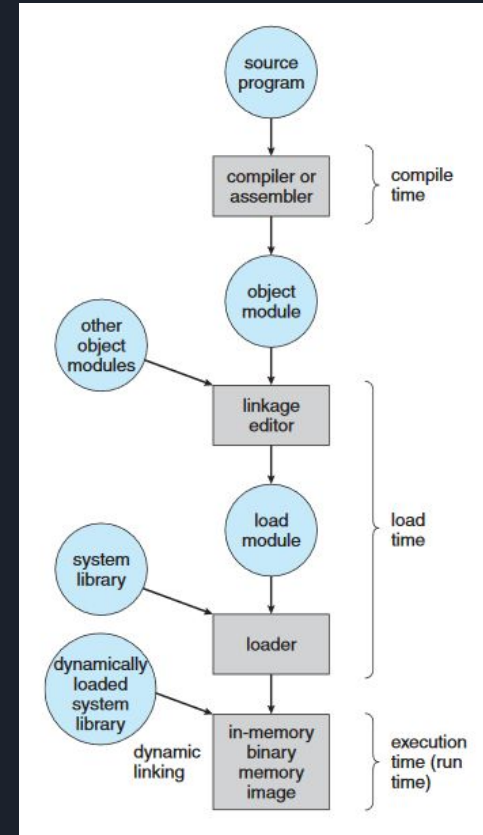
Dynamic Runtime Loading

- bindet erst kurz vor Ausführung
- lädt die direkte physische Adresse
- Code kann vor Ausführung ohne Konflikte geändert werden



Dynamic Runtime Loading (2)

- Programm wird kompiliert
- externe Objekte werden gelinkt
- externe Bibliotheken werden geladen
- während Ausführung werden verwendete Adresswerte berechnet





Position Independent Code (PIC)

- Code, der nicht absolut sondern relativ zum Programm Counter angegeben wird

Vorteile:

- kann ohne Konflikte in den Speicher anderer Programme geladen werden

Nachteile:

- erfordert mehr Ladezeit für jede globale Variable und jeden Funktionsaufruf

- ein Register für GOT Adresse blockiert (RBX) -> minimal langsamere Laufzeit



Global Offset Table (GOT)

- wird bei call von Funktionen vom Loader verwendet
- beinhaltet Offset Werte von shared libraries
- gibt absolute Adresswerte zurück an Loader
- liegt im Data-Segment (im Text-Segment müsste je Referenz relocated werden)

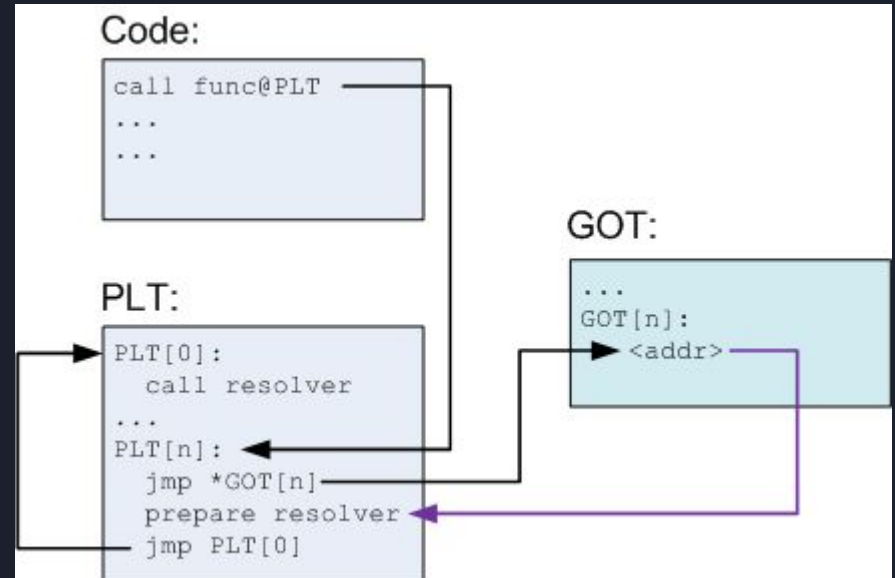


Lazy Binding

- normalerweise werden benötigte Libraries komplett gebunden
- Lazy Binding bindet erst, wenn Funktion im Programmfluss aufgerufen wird
- vermeidet überflüssiges Binding (z.B. Error-Handling)
- erfolgt über PLT

Procedure Linkage Table (PLT)

- bei Funktionsaufruf wird PLT gecallt
- "trampoline"
- PLT berechnet absolute Adresse aus GOT



<https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/>



Address Space Layout Randomisation (ASLR)

- Sicherheitsmaßnahme, die Layout des virtuellen Speichers randomisiert
- Startadressen von Stack, Heap, Libraries, Text/Data-Segment werden zufällig gewählt
- „per process randomisation“
- schützt vor Buffer-Overflow-Angriffen auf z.B. Rücksprungadresse oder Stack



Posix Signale

- Werden von Prozess zu Prozess gesendet

- Integers

- Führen zu Statusänderungen beim Empfänger-Prozess

- Beispiel: SIGSEV -> Benutzung einer Speicherstelle, die nicht vom Prozess reserviert wurde



Signal Handler

- Signalbehandlung manuell möglich

- Linux Debugger GDB SignalHandler : *"info signal"*

- `sighandler_t signal(int signum, sighandler_t handler);`



Beispiel Signalbehandlung in C

```
int main()
{
    int err = signal(SIGINT, sig_handler);
    if( err == SIG_ERR)
        exit(EXIT_FAILURE);
    sleep(3);
    return EXIT_SUCCESS;
}

void sig_handler(int sig)
{
    printf("Caught Signal\n");
    exit(EXIT_SUCCESS);
}
```



Fragen?





sys_ptrace

- sys_ptrace ist ein Systemaufruf unter Linux

- Tracer übernimmt Kontrolle eines Tracee (parent and child Beziehung)



Ptrace in C

```
long ptrace (                \\ Rückgabewert
enum _ptrace_request request , \\ PTRACE - Variante
pid_t pid ,                 \\ Prozess Id des Tracee
void * addr ,               \\ Pointer eines Offset
void * data ,               \\ Pointer auf eine Speicherstelle
);
```



Ptrace -Varianten

-PTTRACE_PEEKTEXT

-PTTRACE_POKETEXT

-PTTRACE_SYSCALL

-PTTRACE_CONT

-PTTRACE_GETSIGINFO



Initialisierung von Tracer und Tracee

-Zwei Möglichkeiten:

-PTRACE_ATTACH: Initialisierung durch Tracer

-PTRACE_TRACEME: Initialisierung durch Tracee



PTRACE_ATTACH

```
-ptrace(PTRACE_ATTACH, tracee_pid, NULL, NULL);
```

-Tracee erhält ein SIG_STOP

Vorteil:

- Anwendung bei bereits laufendem Programm möglich



PTRACE_TRACEME

```
pid_t pid = fork();

if(pid < 0)
    exit(EXIT_FAILURE);

if(pid)
{
    waitpid(pid, 0, 0);
    ...
}
else
{
    ptrace(PTRACE_TRACEME, 0, 0, 0);
    execvp(target[0], target);
    exit(EXIT_FAILURE);
}
```



Initialisierung und nun?

- ptrace Aufruf mit unterschiedlichen Parametern

- Der Tracee ist gestoppt. Fortsetzen mit :

```
ptrace( PTRACE_CONT, tracee_pid, 0, 0 );
```

(Sendet ein SIGCONT)



Ende der Tracer - Tracee Beziehung

Unabhängig vom Initialisierungsvorgang:

- Tracee Programm wird vom User beendet
(zum Beispiel durch den Benutzer mit SIGKILL)
- Tracee beendet sich selbst

Bei PTRACE_ATTACH:

```
-ptrace( PTRACE_DETACH, tracee_pid, 0, 0)
```

Vorteil: Tracee Programm kann weiter laufen



Kernaufgaben des Debuggers

- Speicher auslesen und verändern
- Code injizieren
- Breakpoints setzen
- Singlestep
- Signale senden
- Stackframe anzeigen



Speicheradressen auslesen

- mit `PTRACE_PEEKUSER` werden Register des Tracee ausgelesen
(Zugriff auf USER area Segment)

- mit `PTRACE_PEEKDATA` und `PTRACE_PEEKTEXT` werden Adressen und Register des Tracee ausgelesen

```
uint64_t value = ptrace(PTRACE_PEEKTEXT, pid, addr, 0);
```

- mit `PTRACE_GETREGS` werden alle Register auf einen struct geladen

```
ptrace(PTRACE_GETREGS, pid, 0, &struct);
```



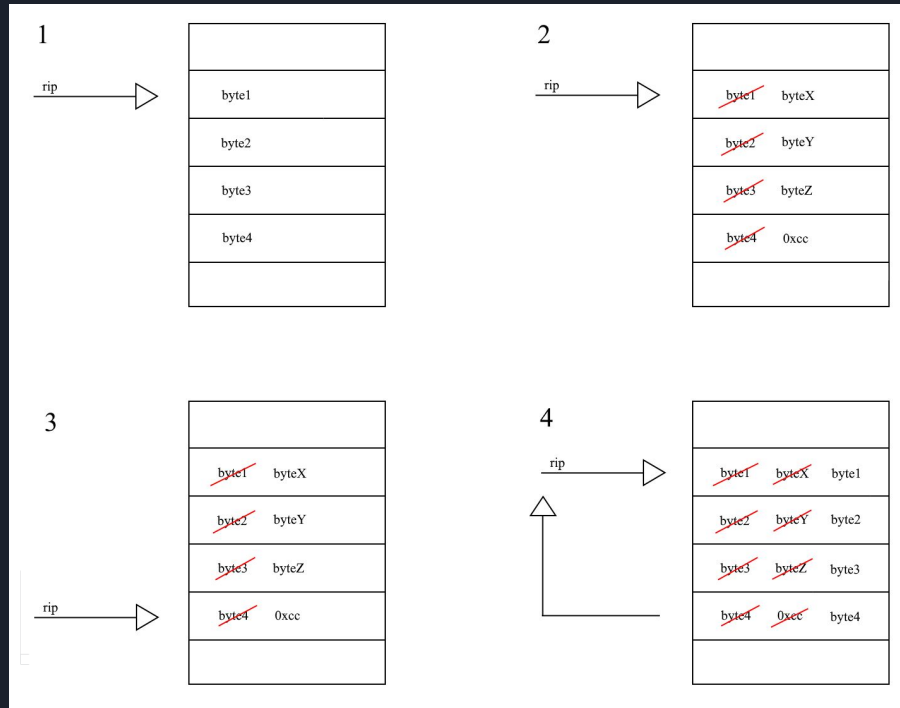
Speicheradressen verändern

- mit `PTRACE_POKEUSER` können Register des Tracee verändert werden

- mit `PTRACE_POKEDATA` und `PTRACE_POKETEXT` können Adressen und Register verändert werden

```
ptrace(PTRACE_POKETEXT, pid, addr ,long_val);
```


Code injizieren





Breakpoints setzen

- Sonderfall der Code-Injizierung, da nur ein Byte überschrieben wird
- mit dem Softwareinterrupt "0xcc" (int3) wird Programmfluss unterbrochen

Breakpointfunktion

```
uint8_t trapcode[8] = {0xcc, 0x90, 0x90, 0x90  
| | | | | 0x90, 0x90, 0x90, 0x90}; // Padding mit NOPs auf 64Bit  
backup = ptrace(PTRACE_PEEKDATA, pid, addr_ptr, 0); // Backup erstellen  
ptrace(PTRACE_POKEDATA, pid, addr_ptr, trapcode); // Trapcode hineinladen  
ptrace(PTRACE_CONT, pid, 0, 0); // Tracee Trapcode ausführen  
wait(0); // Warten auf Breakpoint  
ptrace(PTRACE_POKEDATA, pid, addr_ptr, backup); // Backup laden  
ptrace(PTRACE_POKEUSER, pid, RIP, addr_ptr); // RIP zurücksetzen
```



Probleme durch ASLR

- Zieladressen der Breakpoints durch Begutachten des Maschinencodes gewinnen
- Durch ASLR sind diese zur Laufzeit verschieden!
- Lösung durch Deaktivierung von ASLR?
- Lösung mit Hilfe von `/proc/pid/maps`?
- Lösung durch Vermeiden von PIC?



Probleme durch ASLR - Deaktivierung ASLR

```
echo '0' > /proc/sys/kernel/randomize_va_space
```

- schnelle Lösung

- Sicherheitsrisiko



Probleme durch ASLR - /proc/pid/maps

```
sudo echo /proc/pid/maps
```

- File enthält Informationen zum virtuellen Speicher

- Adressen können errechnet werden

- ASLR kann voll verwendet werden

Probleme durch ASLR - /proc/pid/maps (2)

Adresse	Offset	Fileindex	
7f62e4800000-7f62e49e7000	r-xp 00000000 00:00 18967	/lib/x86_64-linux-gnu/libc-2.27.so	
7f62e49e7000-7f62e49f0000	---p 001e7000 00:00 18967	/lib/x86_64-linux-gnu/libc-2.27.so	
7f62e49f0000-7f62e4be7000	---p 000001f0 00:00 18967	/lib/x86_64-linux-gnu/libc-2.27.so	
7f62e4be7000-7f62e4beb000	r--p 001e7000 00:00 18967	/lib/x86_64-linux-gnu/libc-2.27.so	
7f62e4beb000-7f62e4bed000	rw-p 001eb000 00:00 18967	/lib/x86_64-linux-gnu/libc-2.27.so	
7f62e4bed000-7f62e4bf1000	rw-p 00000000 00:00 0		
7f62e4c00000-7f62e4c26000	r-xp 00000000 00:00 18943	/lib/x86_64-linux-gnu/ld-2.27.so	
7f62e4c26000-7f62e4c27000	r-xp 00026000 00:00 18943	/lib/x86_64-linux-gnu/ld-2.27.so	
7f62e4c27000-7f62e4e28000	r--p 00027000 00:00 18943	/lib/x86_64-linux-gnu/ld-2.27.so	
7f62e4e28000-7f62e4e29000	rw-p 00028000 00:00 18943	/lib/x86_64-linux-gnu/ld-2.27.so	
7f62e4e29000-7f62e4e2a000	rw-p 00000000 00:00 0		
7f62e4e80000-7f62e4e82000	rw-p 00000000 00:00 0		
7f62e5000000-7f62e5001000	r-xp 00000000 00:00 12203	/home/lenni/a.out	
7f62e5200000-7f62e5201000	r--p 00000000 00:00 12203	/home/lenni/a.out	
7f62e5201000-7f62e5202000	rw-p 00001000 00:00 12203	/home/lenni/a.out	
7fffe14aa000-7fffe14cb000	rw-p 00000000 00:00 0	[heap]	
7fffe8181000-7fffe8981000	rw-p 00000000 00:00 0	[stack]	
7fffe9058000-7fffe9059000	r-xp 00000000 00:00 0	[vdso]	
Rechte	Device	Pfad	



Probleme durch ASLR - Vermeiden von PIC

- PIC: - Relative Adressierung

- Bibliotheken effizient laden

- Ermöglicht ASLR

```
gcc input.c -no-pie
```

- Ohne PIC wird ASLR unter Linux nicht verwendet

- Bereits als PIC kompilierte Programme nicht vollwertig zu Debuggen



Singlestepper

- Stop nach jeder Instruktion oder Systemaufruf

- Von ptrace unterstützt:

```
ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL); // springt zur nächsten
                                              // Instruktion

ptrace(PTRACE_SYSCALL, pid, NULL, NULL);    // springt in oder aus
                                              // dem nächsten Syscall
```



Intuitiver Singlestepper

- Kombination aus `PTRACE_SINGLESTEP` und `PTRACE_SYSCALL`

1. Möglichkeit:

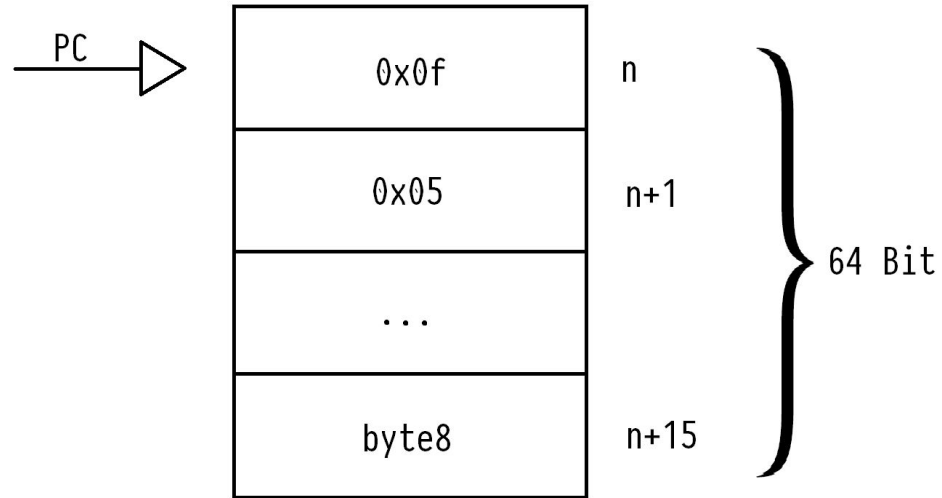
- manuelles Setzen von Breakpoints nach jeder Instruktion und Syscall (`PTRACE_POKEDATA`)

2. Möglichkeit:

- Betrachten des nächsten Opcodes, ob `0x0F 0x05` (Syscall) folgt

- je nach Opcode `PTRACE_SINGLESTEP` oder `PTRACE_SYSCALL`

Intuitiver Singlestepper (2)



Intuitiver Singlestepper (3)

$$\left(\begin{array}{l} 0x-- \dots 0x-- \ 0x05 \ 0x0F \\ \text{AND} \ 0x00 \dots 0x00 \ 0xFF \ 0xFF \end{array} \right) \\ == \ 0x05 \ 0x0F$$

```
if(peek_reg(RIP) & 0xFFFF == 0x050F)
{
    ...
}
```



Tracee erhält ein Signal

- Kernel wählt Thread zum Bearbeiten des Signals aus (Ausnahme Signal = SIGKILL)
(wenn Prozess kein Tracee ist !)
- Tracee wird in *signal-delivery-stop Mode* versetzt
- Tracer hat nun Kontrolle (Signal wurde noch nicht zum Tracee geschickt)
- Signalhandler:

```
ptrace(PTRACE_GETSIGINFO, 0, siginfo_t)
```



Tracer sendet Signale

-Signal weiterleiten oder unterdrücken?

Lösung: Wie macht es der GDB -> info signal

-Signale senden mit `ptrace(PTRACE_CONT, pid, 0, sig)`

`sig` ist Integer vom Signal

(Ausnahme "`sig = 0`" -> kein Signal wird weitergeleitet)



Stackframe anzeigen

- auf Stack liegen gepushte Elemente wie z.B. lokale Variablen, Rücksprungadresse
- Betrachten des Stacks hilft bei z.B. Buffer Overflow Angriffe auf Stack
- Grenzen des Stacks werden mit RSP und RBP ermittelt
- RSP unterste Adresse
- RBP -8 ist oberste Adresse

Stackframe anzeigen (2)

Nach ENTER:

	ebp + 12
esp + 8	ebp + 8
esp + 4	ebp + 4
esp	ebp

...
Parameter high
Parameter low
Rücksprungadresse
gerettetes ebp

Sysprog 2019 Kapitel 1.0
(Maschinenprogrammierung)

```
# enter
push RBP
mov RBP, RSP
```

```
# leave
mov RSP, RBP
pop RBP
```




'DefectDeflect' Debugger

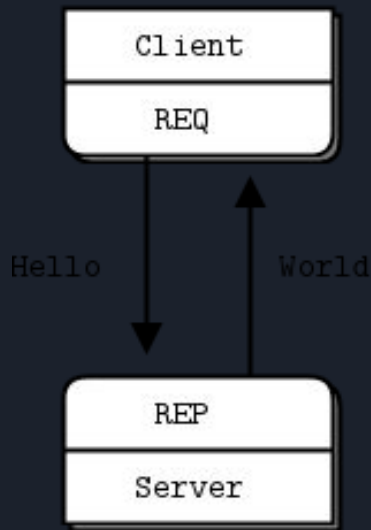
- Unterteilung in Server und Client
- > Modularität
- Server: Low Level Implementierung mit ptrace
- Client: abstraktere Debugging -Funktionen über Schnittstelle mit Server
- ZMQ und API zur Kommunikation



ZMQ - Zero Message Queue

- Bibliothek für Nachrichtenaustausch
- Viele Schnittstellen für verschiedene Programmiersprachen
- Statt Berkeley Sockets
- Einfache Handhabung
- Bloat

ZMQ - Zero Message Queue (2)



```
// create 'ping pong client' socket
zsock_t* client = zsock_new(ZMQ_REQ);
// error check
assert_not_null(client);
// connect to server at localhost
int err = zsock_connect(client, "tcp://127.0.0.1:5555");
// error check
assert_no_err(err);
```

ZMQ - Zero Message Queue (3)



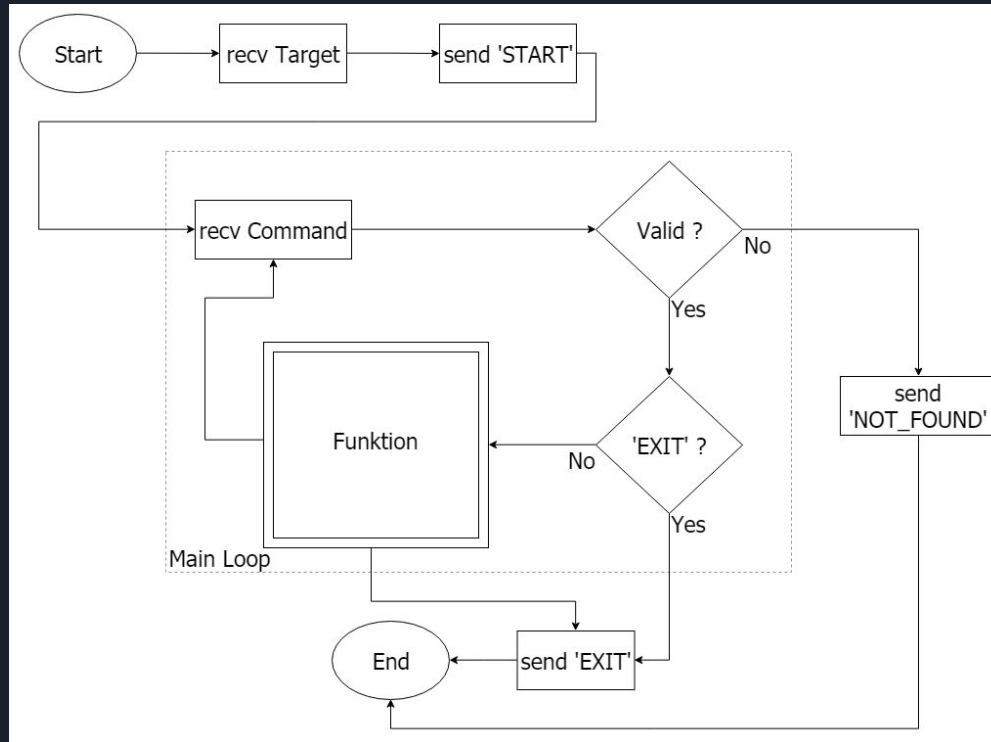
```
// sending a string
zstr_send(client, "Hello, Server!");
char* response;
// receive a string
response = zstr_recv(client);
puts(response);
free(response);
// destroy the socket
zsock_destroy(client);
```



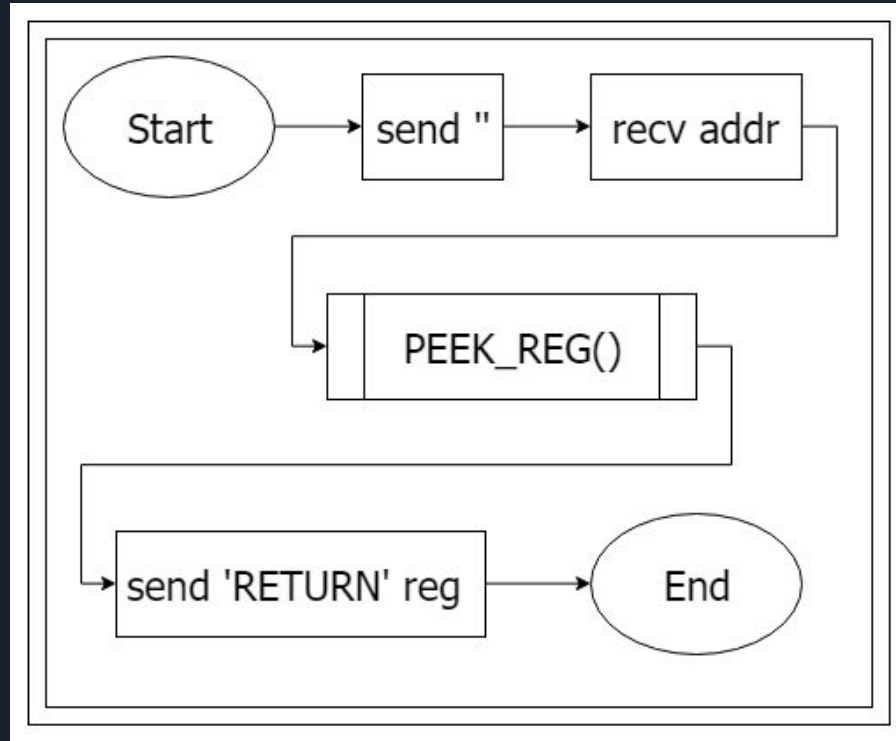
Server Pseudocode

```
network_init();           // Einrichten der IPC
tracee_init();            // Start des Tracee und PTRACE_TRACEME
loop:
    receive_command();     // Erhalten des Befehls über IPC
    execute_command();     // Durchführen des Befehls unter Zuhilfenahme von sys_ptrace
goto loop;
destroy();                // Beenden des Tracee und der IPC
```

API Flowchart



API Flowchart - PEEK_REG





Erweiterbarkeit

- Um GUI / CLI erweitern
- Um API Funktionalitäten erweitern

Erweiterbarkeit (2)

Implementieren

```
void __my_function__(pid_t pid)
{
    ptrace(...);
}
```

API definieren

```
void func_my_function(zsock_t* sock, pid_t pid)
{
    __my_function__(pid);
    zstr_send(sock, "RETURN");
}
```

Mappen

```
func["MY_FUNCTION"] = &func_my_function;
```

Deklariieren

```
void func_my_function(zsock_t* sock, pid_t pid);
void __my_function__(pid_t pid);
```



Command Line Input

- Bedienung für Debugger durch I/O
- in C geschrieben und mit ZMQ realisiert
- Funktionalität > Komfort
- Einfache Erweiterbarkeit für neu implementierte Funktionen
- Optimierbar durch: z.B.: automatisierte Registerausgabe, bessere Visuelle Darstellung,



Ausblick und Planung

- Disassemblieren
- Auflösen von Symbolen
- Parsen von Opcodes
- Step in und Step over



Step in und Step over

- Behandlung von Funktionsaufruf
- Step in setzt Breakpoint in aufgerufene Funktion an erste Stelle
- Step over überspringt Funktionsaufruf und setzt Breakpoint nach der return address



Step in und Step over (2)

```
void main()
{
    foo();    // <- RIP aktuell
    bar();    // <- RIP nach 'Step over'
}

void foo()
{
    baz();    // <- RIP nach 'Step in'
    qux();
    return;
}
```



Fragen?





Create Remote Thread Windows

- erstellt einen Thread in VM eines Ziel-Prozesses
- Thread hat zugriff auf alle Objekte die vom Prozess geöffnet werden
- Jugaad ist Linux 32 Bit Implementierung von Create Remote Thread
- Ziel: Jugaad nach 64 Bit portieren

Create Remote Thread Windows

```
1  HANDLE CreateRemoteThread(  
2      HANDLE                hProcess,  
3      LPSECURITY_ATTRIBUTES lpThreadAttributes,  
4      SIZE_T                dwStackSize,  
5      LPTHREAD_START_ROUTINE lpStartAddress,  
6      LPVOID                lpParameter,  
7      DWORD                 dwCreationFlags,  
8      LPDWORD               lpThreadId  
9  );
```




-hProcess Ein Handle des Threads mit Security und Access Rights

-lpThreadAttributes

Pointer zu Security Attributes spezifiziert einen security descriptor

dwStackSize wird auf eine Pagegröße aufgerundet

lpStartAddress ein Pointer zum Threadcode

lpParameter Parameter die dem Thread übergeben werden können


dwCreationFlags immediately Start / suspended state

lpThreadId Ein Pointer einer Variable- erhält den Thread identifier



Jugaad


```
1  int create_remote_thread_ex(pid_t pid,  
2                               size_t stack_size,  
3                               unsigned char *tpayload,  
4                               size_t tpsize,  
5                               int thread_flags,  
6                               int mmap_prot,  
7                               int mmap_flags,  
8                               void * bkpaddr)
```



pid	Nummer des Prozess in den der Thread injeziert wird
stack_size	Größe des Stacks
*tpayload	pointer auf eine Payload der injiziert wird
tpsize	länge der Payload
thread_flags	Handle des Prozess ->(clone flags)
mmap_prot/flags	Flags für Access Right(mmap flags)
*bkpaddr	Zieladresse der Injektion

C-mmap funktion speicher allokkieren

```
1  void *mmap(void *addr,  
2           size_t length,  
3           int prot,  
4           int flags,  
5           int fd,  
6           off_t offset);
```

- 
- *addr pointer eine präferenzierten Adresse
 - Size_t größe des zu reservierenden Speichers
 - Prot Parameter sicherheitsrelevanten Eigenschaften -> Access Rights
 - Flags entscheidet über Anonymous / Shared des mapping
 - fd ein möglicher File Descriptor
 - Offset offset einer File, geöffnet im fd



Flags und Prots

- PROT_EXEC/PROT_READ/PROT_WRITE

- MAP_ANONYMOUS:- Mapping wird mit 0 initialisiert

 - fd und offset werden ignoriert

 - kein verweis auf das neue mapping

- MAP_PRIVATE: -Erstellt ein privates copy-on-write mapping


 - Änderung für andere Mappings der gleichen

File nicht sichtbar



C-Clone Funktion Thread erstellen

```
1  int clone(int(*fn)(void *),  
2          void *child_stack,  
3          int flags,  
4          void *arg)
```



`int(*fn)(void *)` function pointer der Thread Funktion

`Void *child_stack` Threadstack (wächst nach unten)

`Int flags` Bestimmt Signalhandling/ThreadGroup ...

`Void *arg` Argumente, die an den Thread weitergegeben werden

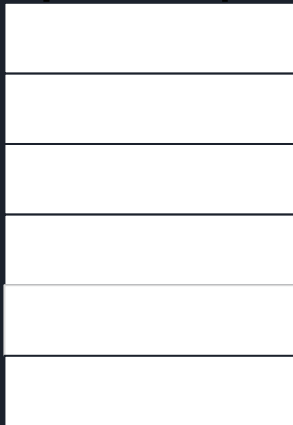


Clone Flags

- CLONE_THREAD: -Thread gehört selben Thread Group des Callers
 - Thread hat den selben Parent
 - beim terminieren wird kein SIGCHILD gesendet
- CLONE_SIGHAND & CLONE_VM muss gesetzt werden
- CLONE_SIGHAND: -Teilt den Signalhandler mit dem Caller
- CLONE_VM: -Teilt den selben VM mit dem Caller



Prozess VM



PTRACE_ATTACH




Jugaad

SHELLCODE 1

SHELLCODE 2

SHELLCODE 3



Prozess VM



PTRACE_PEEKDATA



Jugaad

BACK_UP

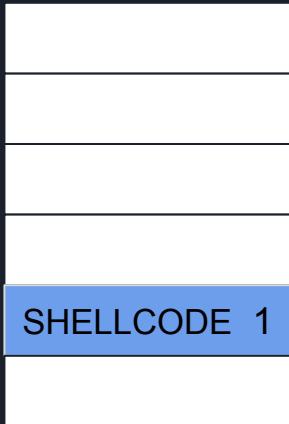
SHELLCODE 1

SHELLCODE 2

SHELLCODE 3



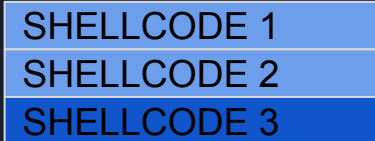
Prozess VM



← PTRACE_POKETEXT

Jugaad

BACK_UP

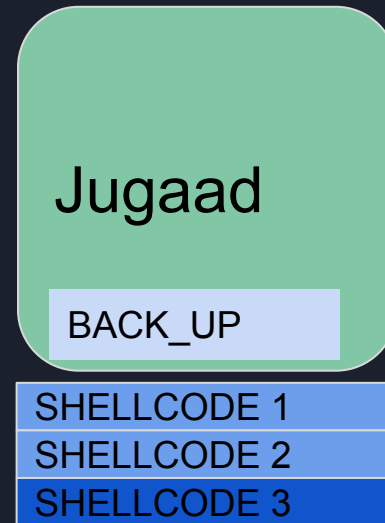
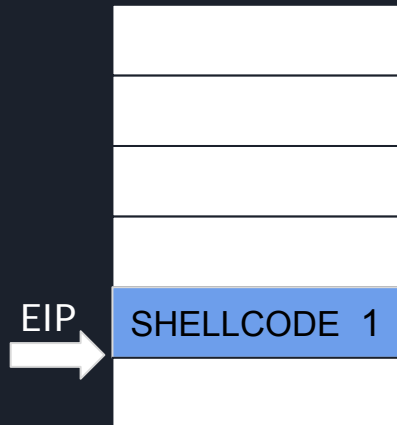


SHELLCODE 1
SHELLCODE 2
SHELLCODE 3



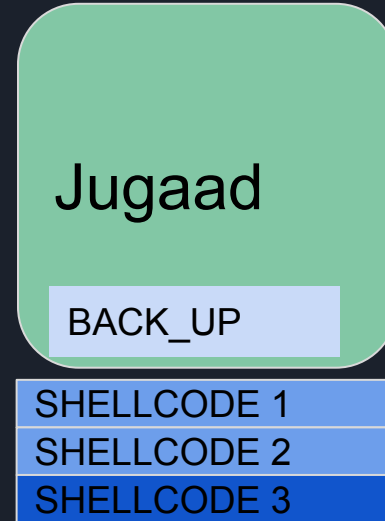
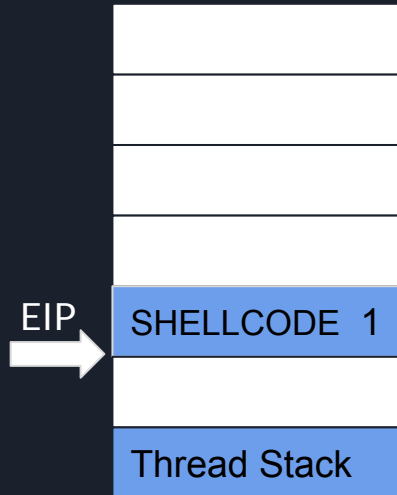


Prozess VM

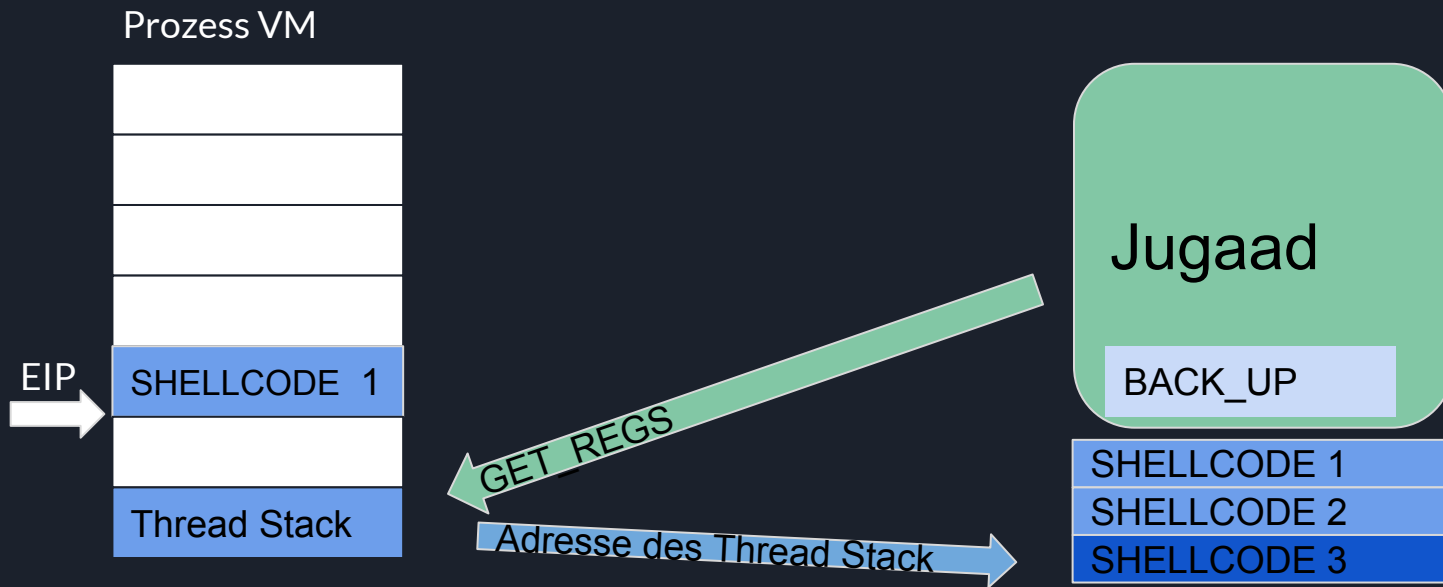


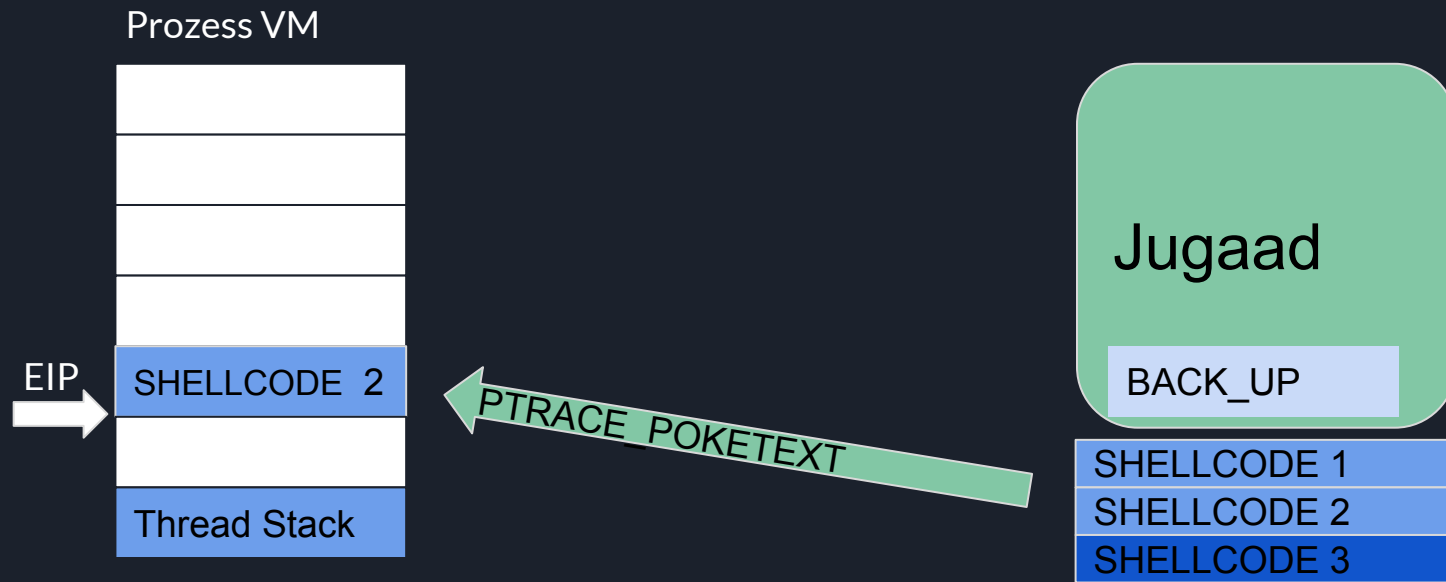


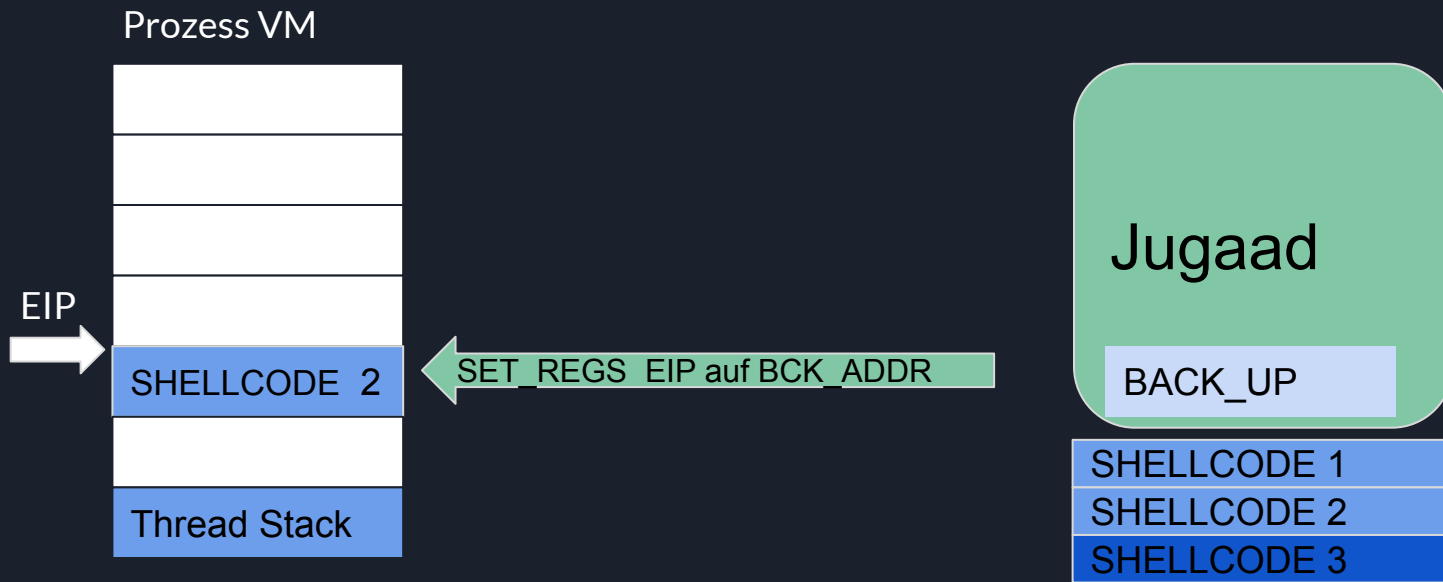
Prozess VM





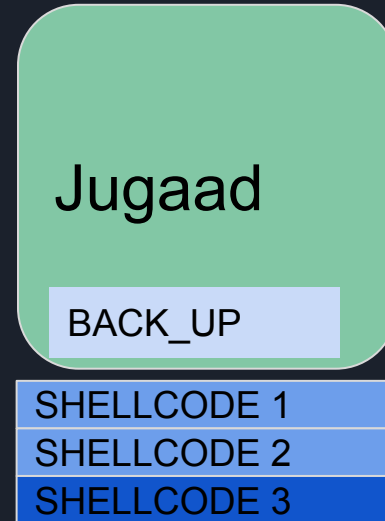
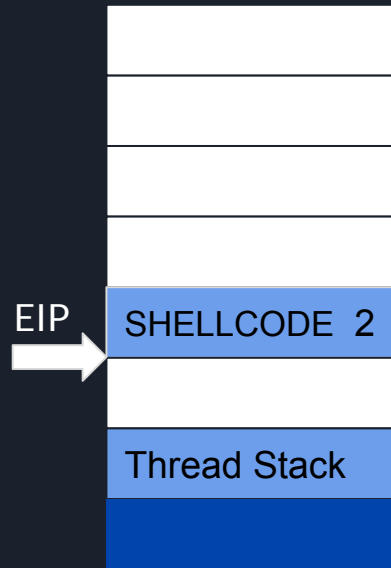


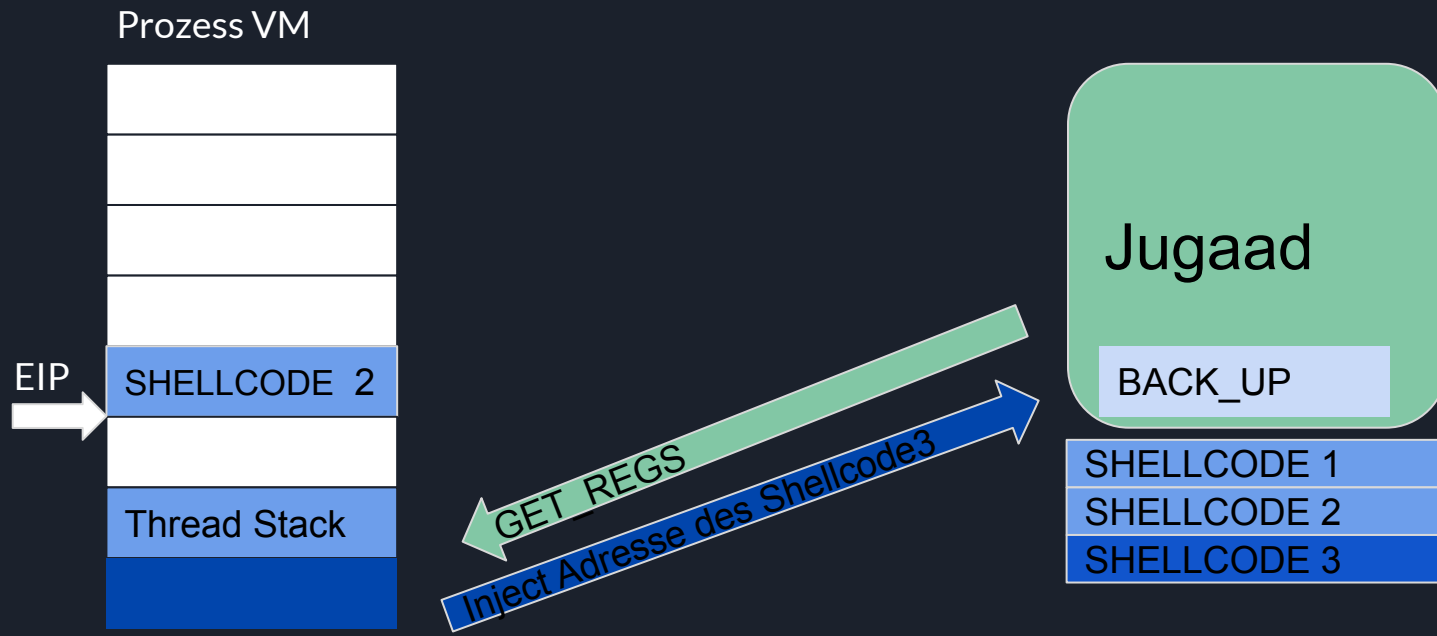






Prozess VM



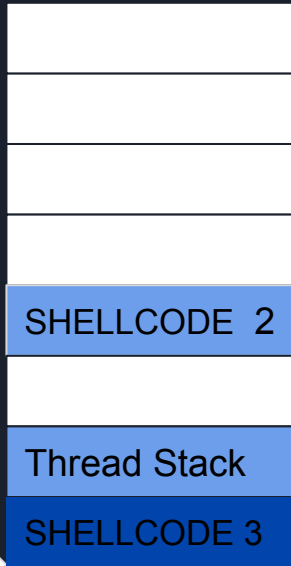








Prozess VM



EIP



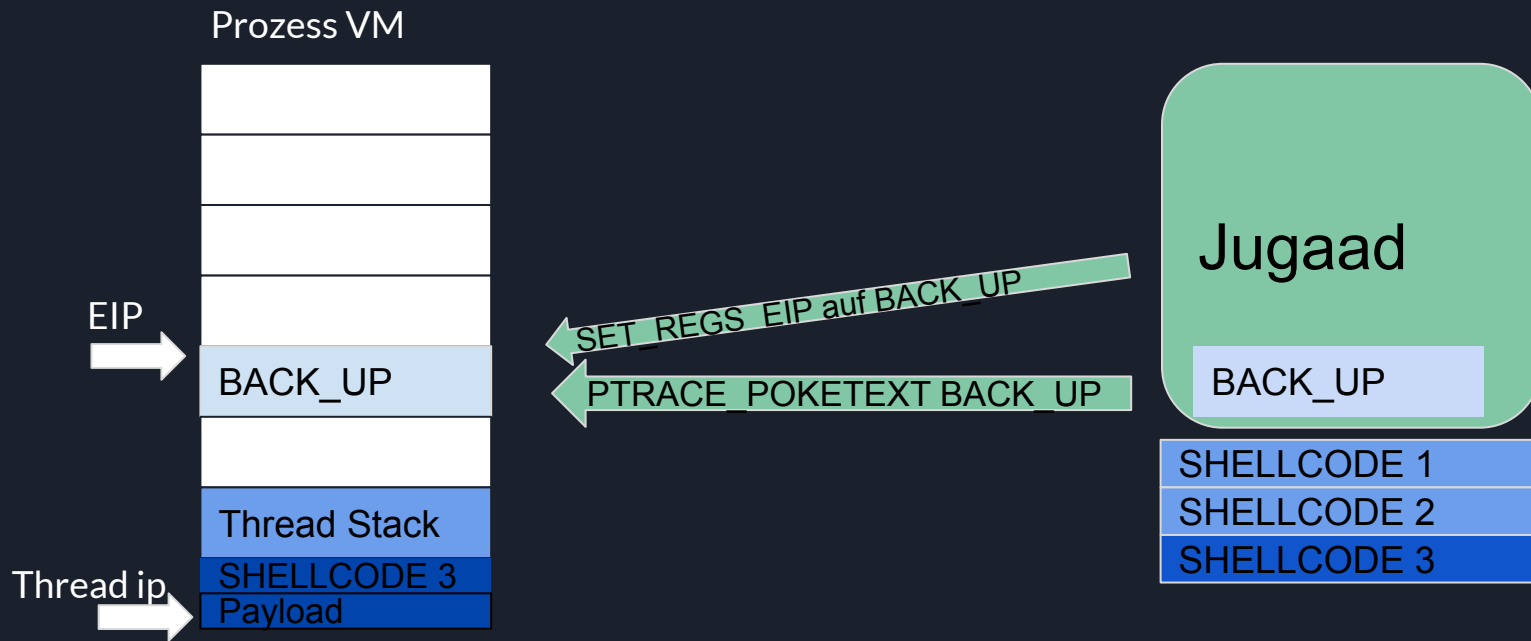
Jugaad

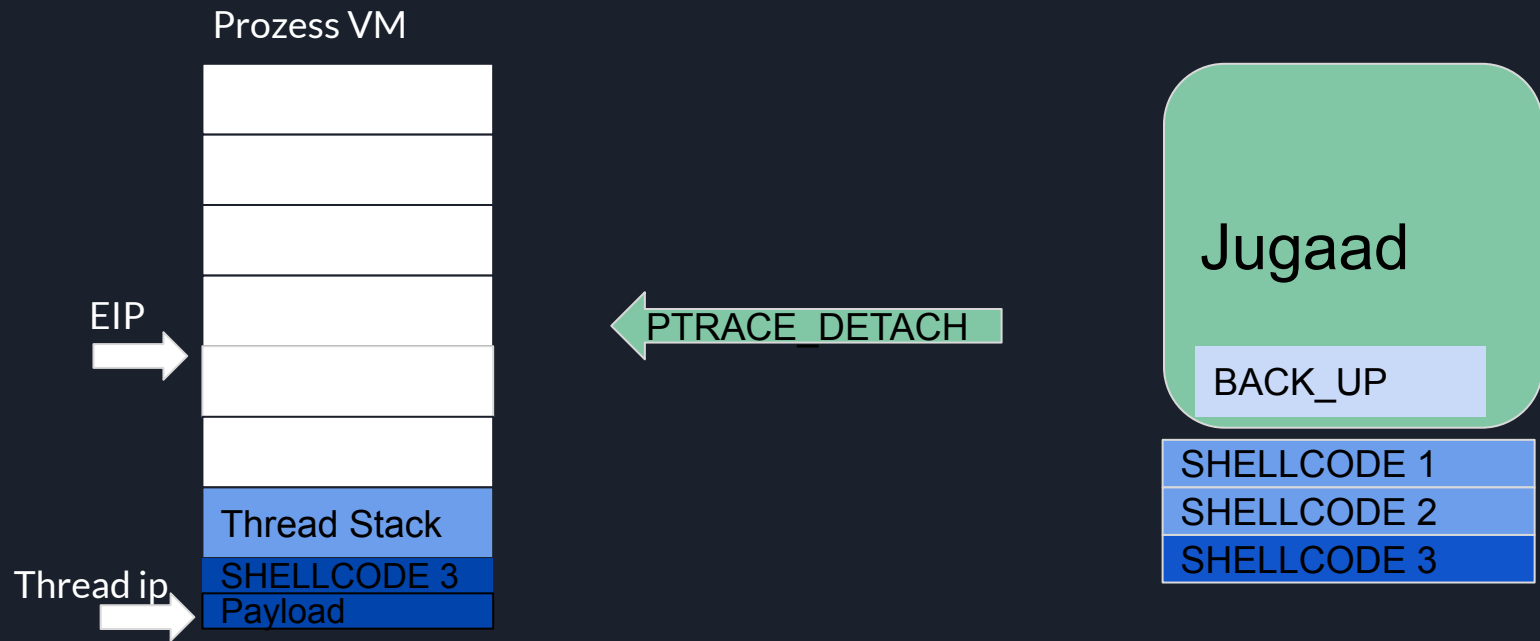
BACK_UP

SHELLCODE 1

SHELLCODE 2

SHELLCODE 3







64-Bit Implementierung

- Jugaad : ptrace zugriff auf Registern müssen verändert werden

 - >EAX zu RAX, EIP zu RIP

- ptrace Code Injektion müssen auf `size_long` angepasst werden
im 64 bit System sind das 64bits

- Shellcodes müssen auf 64bit neu umgeschrieben werden



SHELLCODE 1

```
1  mov rdi,0           //void* addr
2  mov rsi,STACK_SIZE  //Die gröÙe des Stacks haben wir auf 4096*1026 gesetzt (4Mb)
3
4  mov rdx,PROT_EXEC|PROT_READ|PROT_WRITE
5  mov r10,MAP_PRIVATE|MAP_ANONYMOUS
6
7  mov rax,9           //Syscall Nummer von Sys_mmap()
8  syscall
9  int3               //Debugging interrupt Signal
```

SHELLCODE 2 & 3 Verknüpft

```
1  int payload(void *arg)
2  {
3      do stuff ....
4      exit(0);
5  }
6  void shellcode(int (*fn)(void *)){
7      int *stack_pointer = mmap(NULL, Stacksize, Prots, Flags, -1, 0);
8      clone(fn, stack_pointer+Stacksize, Flags, NULL);
9  }
10 int main()
11 {
12     shellcode(payload);
13     return 0;
14 }
```



X86_64 für Clone


Kernel erwartet:

- rax: system call number


- rdi: flags

- rsi: child_stack

->Problem : Wo kommt `(int (*fn)(void*))` hin ?



```
1  thread_create:
2      push rdi
3      call get_stack
4      lea rsi,[rax + STACK_SIZE -8]
5      pop qword[rsi]
6      mov rdi, CLONE_THREAD|CLONE_VM|CLONE_SIGHAND
7      mov rax, SYS_clone
8      syscall
9      ret
10 get_stack:
11     ..;mmap shellcode ohne int3
12     ret
```

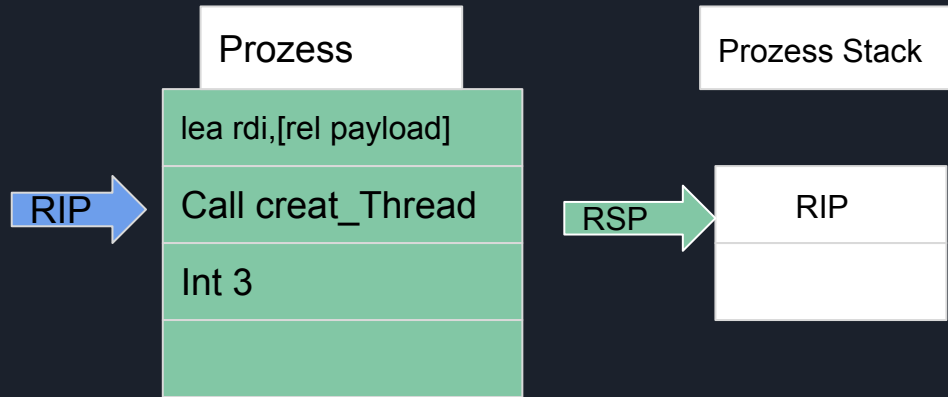


```
1  thread_create:
2      push rdi
3      call get_stack
4      lea rsi,[rax + STACK_SIZE -8]
5      pop qword[rsi]
6      mov rdi, CLONE_THREAD|CLONE_VM|CLONE_SIGHAND
7      mov rax, SYS_clone
8      syscall
9      ret
10 get_stack:
11     ..;mmap shellcode ohne int3
12     ret
```

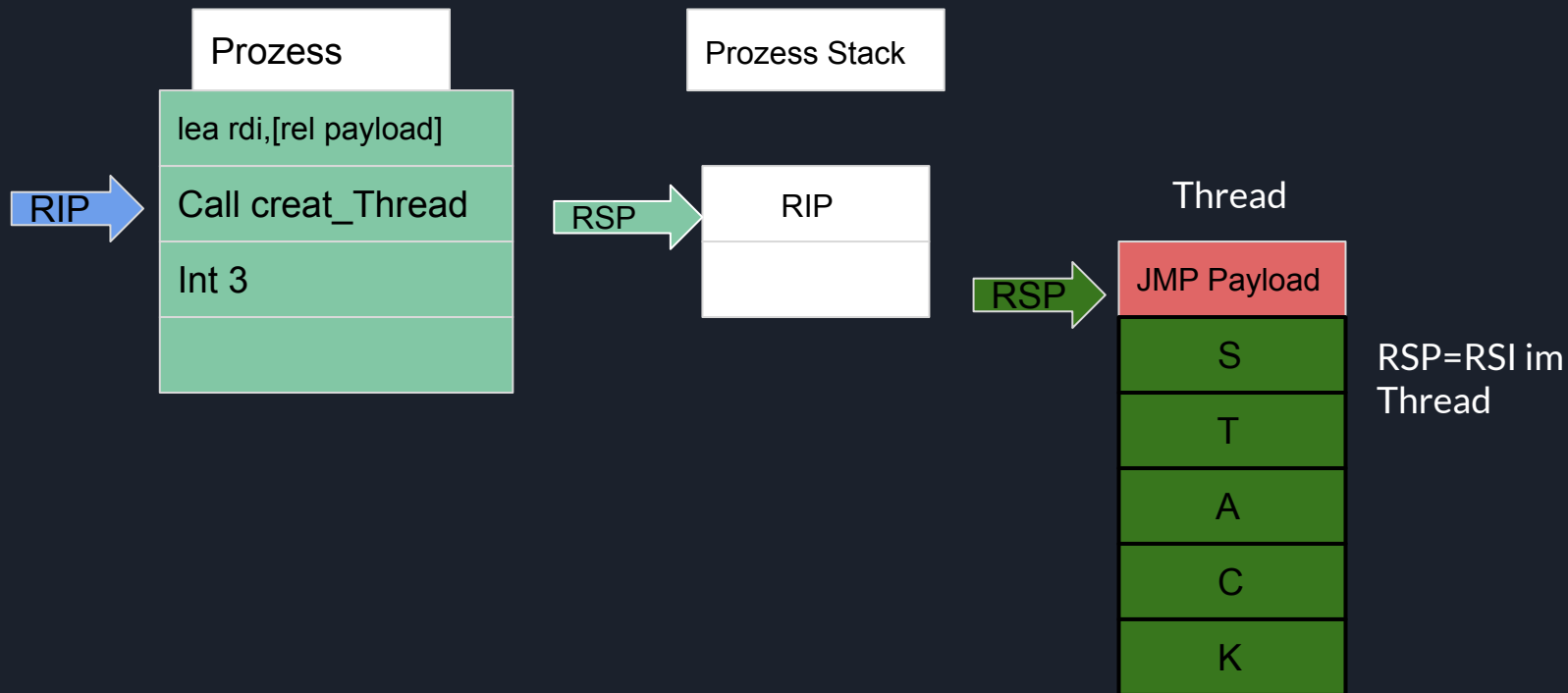
Diagram illustrating the `ret` instruction in the `thread_create` function:

- The `ret` instruction at line 9 is circled in red.
- A red arrow points from the circled `ret` to a green box labeled "Call XY", which contains the instruction `ret`.
- Another green box to the right contains the instructions `push %rip`, `jmp XY`, and `popq %rip`.

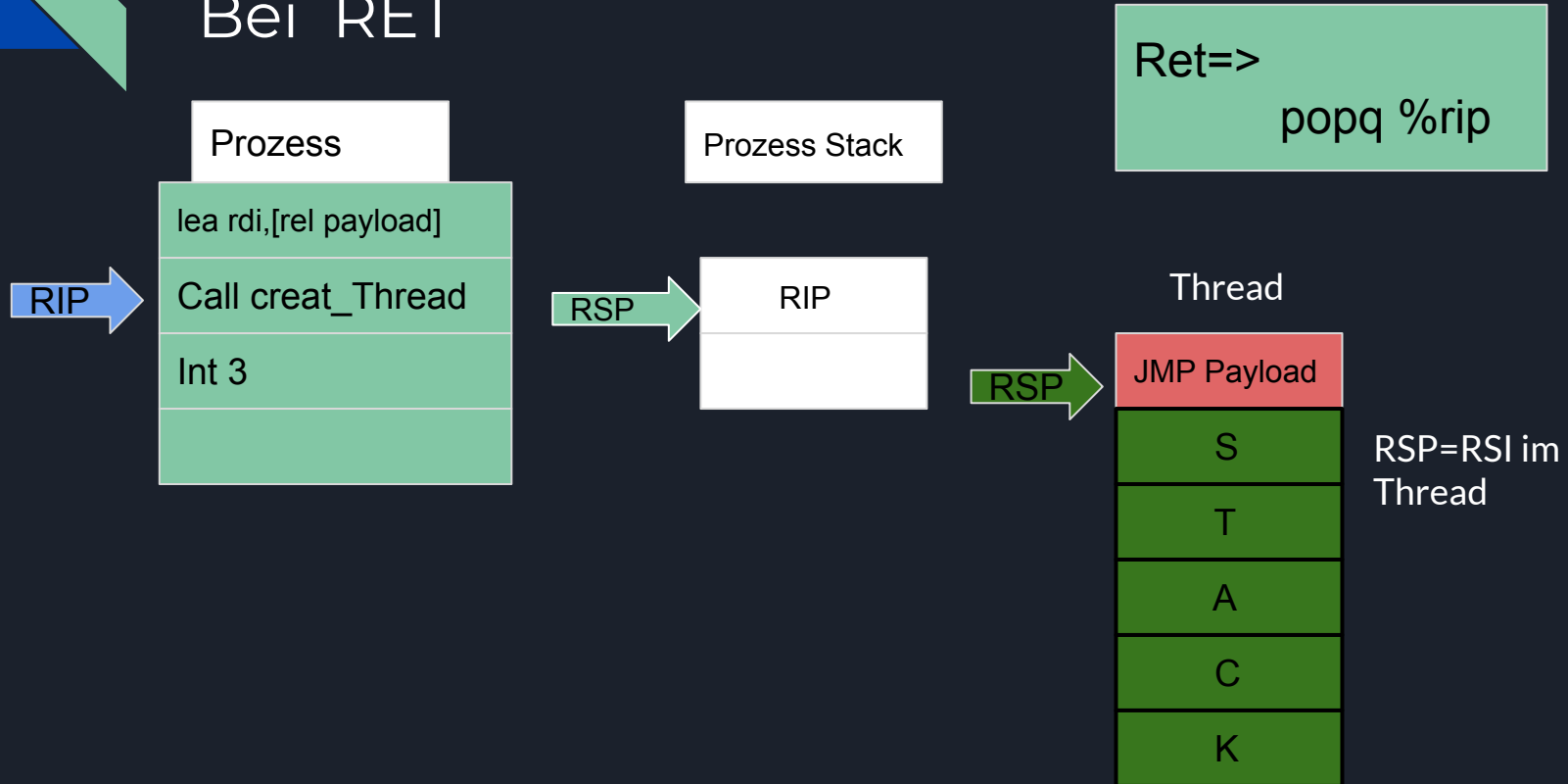
Bei CALL



Bei RET



Bei RET



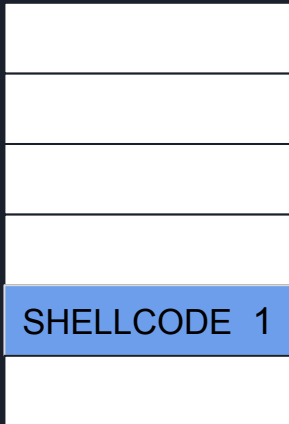


Payload / Thread Funktion

- Bsp "Hello World"
- relative Adressierung
- exit sys_call , damit der Programmfluss nicht gestört wird



Prozess VM



← PTRACE_POKE TEXT

Jugaad

BACK_UP

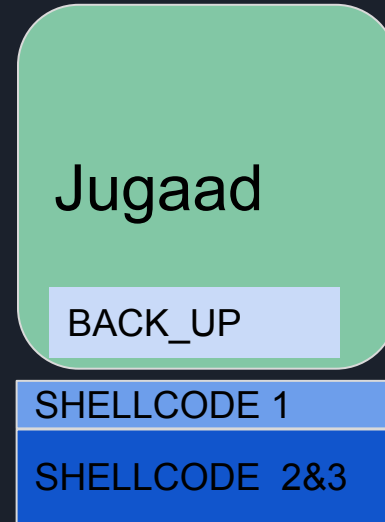
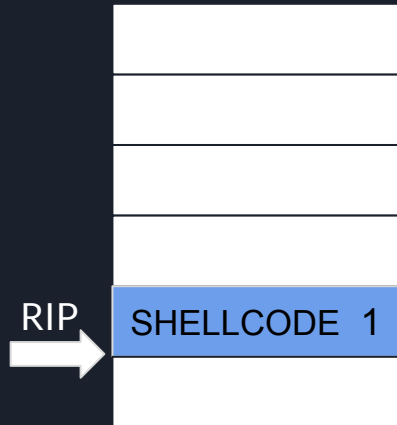
SHELLCODE 1

SHELLCODE 2&3



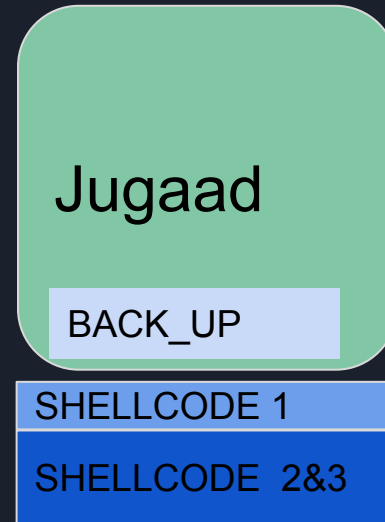
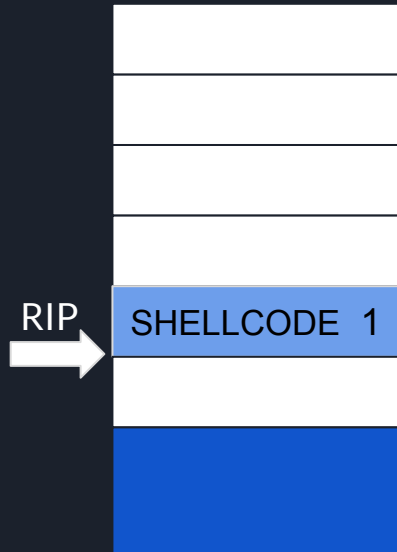


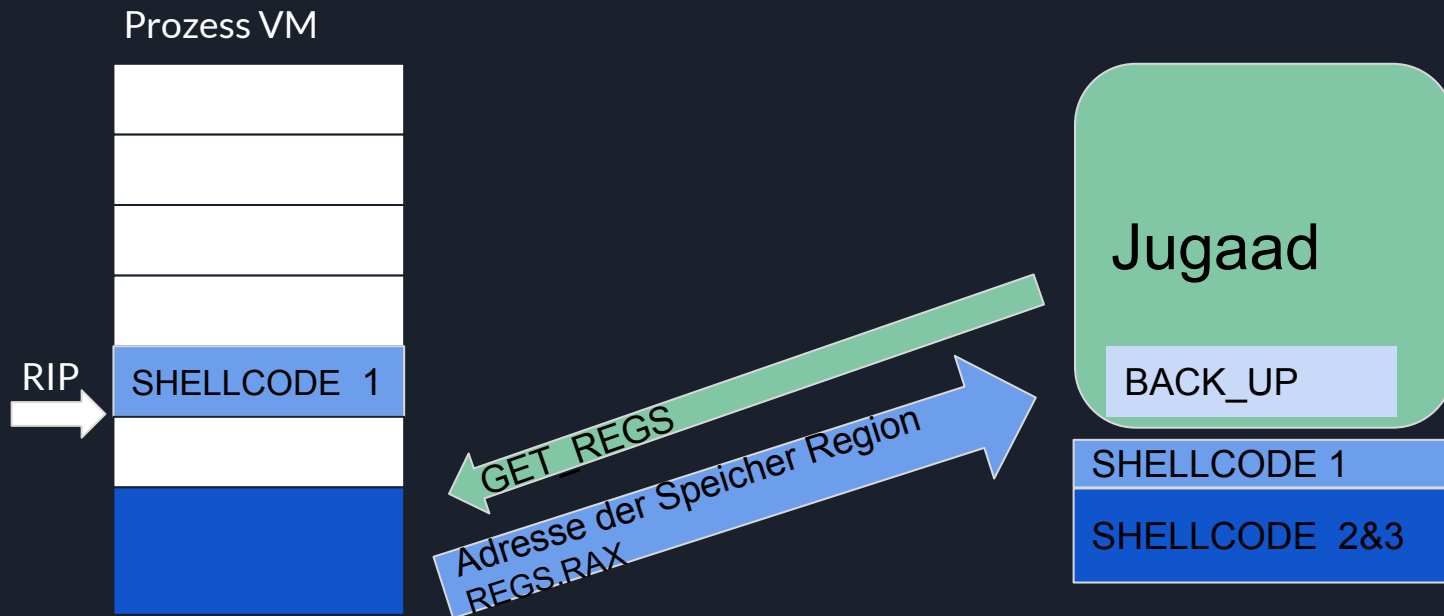
Prozess VM



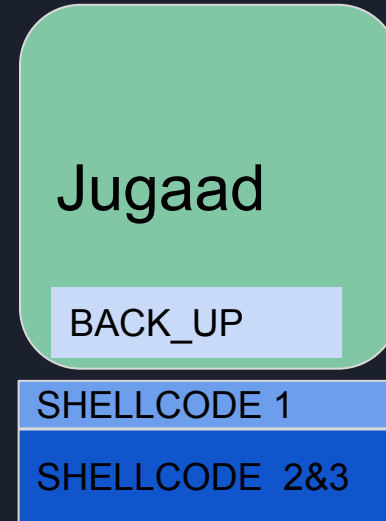
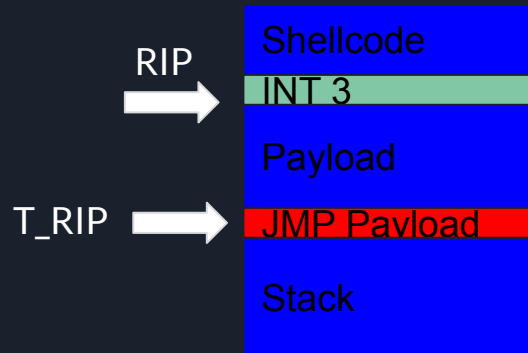


Prozess VM









Prozess VM

RIP



BACK_UP

T_RIP



Shellcode
Payload
JMP Pavload
Stack

SET REGS RIP auf BACK_UP

PTRACE POKETEXT BACK_UP

Jugaad

BACK_UP

SHELLCODE 1

SHELLCODE 2&3



Prozess VM



← PTRACE DETACH

Jugaad

SHELLCODE 1

SHELLCODE 2&3



Fragen ?

